SEPTEMBER 2003

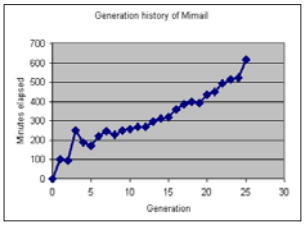
virus BULLETIN

The International Publication on Computer Virus Prevention, Recognition and Removal

CONTENTS

ISSN 0956-9979

IN THIS ISSUE



GENERATION GAP

A bug in Mimail's replication routine allows determination of the time elapsed between virus generations. Gabor Szappanos has all the details of the virus that dethroned Klez.H from the prevalence charts.

page 4

BLASTER CAST

The first Win32 worm to use a command shell attack has arrived, and it is not only the corporate servers that have borne the brunt of the attack – Win32/Blaster demonstrates 'buffer overflow for the masses'. Peter Ferrie, Frédéric Perriot and Péter Ször set out the worm's vital statistics.

page 10

FORMAL CHALLENGE

As researchers in academia and industry begin to develop anti-virus technologies founded on formal methods of analysing programs, Arun Lakhotia and Prabhat K. Singh look at the promises and the pitfalls of formal analysis.

page 15

'Truly recovering data and repairing the file so it is restored to its exact original form is a major philosophical shift in anti-virus architecture.'

Stephen Willis
USA

SALVAGE OPERATION

In September 2001 I was Senior Manager of Worldwide Software Support at a supplier of process control and yield management solutions for the semiconductor and related microelectronics industries. The Nimda and FunLove viruses had been introduced into the market in September 2001 and November 1999, respectively. The corporate standard AV protection within the company was a product from one of the top three anti-virus vendors, and had received the signature file updates within a day or so of the original outbreak. The signature updates were implemented company-wide within minutes of receipt.

Should be OK, right? Unfortunately there was a massive infestation of viruses that was deeply embedded throughout the company. This was the problem: updating signature files was successful in preventing new instances of viruses from being introduced into the company, but the virus was already inside the company. Nimda incorporated new technology designed specifically to defeat the 'detect and quarantine/delete' philosophy of most of today's anti-virus products.

These were the first viruses designed to spread via the IP layer, and they were smart enough to leverage illegal drive mappings and share points to keep spreading back to computers even after they had been cleaned. The number of infected files within the company was enormous. Quarantining or deleting all of this data was unacceptable.

The company was at a point where it potentially needed to cease operations, shut the whole network down, company-wide, and clean every portion of the network completely before being able to bring it back up. Six months had been spent on repair efforts already, and at least $1M dollars in hard costs on the clean-up effort, and the problem was still not resolved. On top of that, the loss of critical data had been huge.

Clearly, the virus author of today does not sit at home, thinking of how to write another virus that can easily be solved by most of the world's anti-virus products of today. They want to create billions of dollars of damage. The extent of the damage is how they will measure their success.

They know today's anti-virus programs, and they study these programs to figure out how to achieve their goal. They are trying to figure out how to beat the current state of anti-virus technology. There is no such thing as a safe solution. And failure to understand this environment can cost an enterprise millions of dollars, and significant losses in market share.

Eventually a solution was found that helped the situation and, over a period of months, the company's virus problem was brought under control. The solution provided the ability to repair files (e.g. critical data) and recover 100 per cent of the content from the files.

Future major viruses will get into the enterprise, and will infect critical data files. The loss of that data will be a huge expense and cost billions of dollars worldwide. The IT landscape is changing constantly, and there are new doorways opening for virus authors – such as web services and wireless handheld devices.

We can't anticipate the doorway that the next major virus author will find. Just know that they will find one. And when they do, the mission-critical task will be to figure out how to recover the data from the files that have been infected.

Truly recovering data and repairing the file so it is restored to its exact original form before the virus infection is a major philosophical shift in anti-virus architecture. It requires a completely different approach and requires tens of man-years of engineering effort to accomplish. It cannot be achieved by re-architecting an existing solution but must be developed from scratch.

The company in this case paid a heavy price to learn this lesson. Let's hope that it comes easier for the rest of us!

NEWS

WATCHING AND WAITING FOR SOBIG

As the final touches were being made to this month's issue of *VB*, anti-virus experts and sysadmins worldwide waited to discover what events, if any, would unfold when Sobig.F began a synchronized attack at 1900h GMT on Friday 22 August 2003 and subsequent second phase of attack at the same time on Sunday. The worm was set to contact 20 predefined IP addresses (encrypted within the virus body) on UDP port 8998, which would then redirect infected machines to a URL from which an unknown program would be downloaded and run.

In the event, the attack failed: all 20 of the IP addresses were inaccessible during the first phase. With reported assistance from the FBI and Royal Canadian Mounted Police, 19 of the 20 servers were taken offline and there were no signs of the virus being able to connect to the one remaining server. Two days later, the second stage of attack was equally unsuccessful: although four of the servers started responding to pings just before the attack began – three of which were listening on port 8998 – none of them responded when sent the worm's eight-byte activation code.

Theories and rumours abound as to the identity of the author of Sobig.F and its predecessors: a number of experts believe the worm is the work of email spammers attempting to build an infrastructure for a spam attack of epic proportion; others believe that a highly organized group of hackers has orchestrated the attack, while it has even been speculated that an individual likened to comic book villain Lex Luther is behind the worm. Unfortunately for would-be superhero AV researchers, all theories currently remain speculation. Meanwhile, a detailed look at the Sobig family is planned for next month's issue of *Virus Bulletin*.

PATENTLY CHANGING HANDS

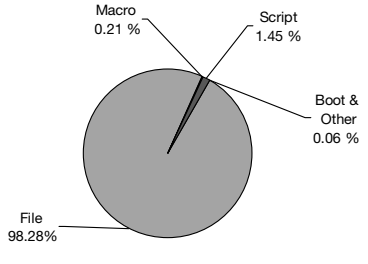
Symantec has paid $62.5m for the infamous US Patent No. 5,319,776, which has been the bane of major players in the anti-virus industry for the last six years. In 1997, software company *Hilgraeve Inc.* filed suits against *Symantec* and *McAfee*, alleging that the companies' email scanners infringed its patent, granted in 1994 (see *VB*, October 1997, p.3). The patent covers searching for virus signatures in data that is being copied between media to inhibit virus infection automatically. Most of the big players in the anti-virus industry have run into the patent at some point – *NAI*, *Trend Micro* and *IBM* settled with *Hilgraeve*, while a number of other companies including *Computer Associates*, *Aladdin Knowledge Systems* and *Clearswift* are still in litigation. *Symantec* is reported to be reviewing which of its competitors are infringing its newly acquired patent, and will consider the possibility of litigation of its own.

Prevalence Table – July 2003

Virus	Type	Incidents	Reports
Win32/Opaserv	File	9064	38.09%
Win32/Sobig	File	5749	24.16%
Win32/Bugbear	File	1799	7.56%
Win32/Dupator	File	1687	7.09%
Win32/Klez	File	1360	5.71%
Win32/Yaha	File	932	3.92%
Win32/Funlove	File	532	2.24%
Win32/Fizzer	File	462	1.94%
Win95/Spaces	File	397	1.67%
Fortnight	Script	167	0.70%
Win32/Kriz	File	142	0.60%
Redlof	Script	131	0.55%
Win32/Deborm	File	128	0.54%
Win32/Ganda	File	125	0.53%
Win32/Mofei	File	111	0.47%
Win32/Lovelorn	File	103	0.43%
Win32/Gibe	File	102	0.43%
Win32/Magistr	File	96	0.40%
Win32/Holar	File	71	0.30%
Win32/Mylife	File	63	0.26%
Win32/SirCam	File	57	0.24%
Win32/Lovgate	File	55	0.23%
Win32/Hybris	File	38	0.16%
Win32/Nimda	File	34	0.14%
Others[1]		394	1.65%
Total		23798	100%

[1]The Prevalence Table includes a total of 394 reports across 70 further viruses. Readers are reminded that a complete listing is posted at http://www.virusbtn.com/Prevalence/.

Distribution of virus types in reports



VIRUS ANALYSIS 1

CHECKING MIMAIL

Gabor Szappanos
VirusBuster, Hungary

I-Worm.Mimail appeared on the first of August 2003. It used *Windows* vulnerabilities to run. Although the worm was not extraordinary in itself, the vulnerabilities it used and the generation data collected on the worm proved interesting enough to warrant a detailed analysis – besides which, any worm that can kick Klez.H off the top of the virus charts deserves a mention.

The worm spreads in an interesting compound sandwich file. It is essentially a MIME file, where the MIME attachment is the worm binary in straight binary storage. This means that, in the message, the worm is not BASE64 encoded, as we are used to.

Appended to this MHTML file is a short JavaScript dropper code. All of this arrives at the potential victim as a ZIP attachment containing an HTML message. When the HTML file is opened, *Microsoft*'s HTML parser will skip the lengthy kilobytes of the binary worm, find the script and execute it. The script will extract and execute the binary worm. The binary worm then repackages itself and spreads to new targets.

SCRIPT COMPONENT

The script component makes use of two known *Windows* vulnerabilities that are described in *Microsoft* security bulletins *MS02-15* and *MS03-14*.

The first is the 'Local Executable Invocation via Object tag' vulnerability, which (according to the security bulletin) allows the execution of any file on the local file system, with the following restrictions:

- The vulnerability would not enable the attacker to pass any parameters to the program.
- An attacker could only execute a file on the victim's local machine. The vulnerability could not be used to execute a program on a remote share or website.
- The vulnerability would not provide any way for an attacker to put a program of his choice onto another user's system.
- An attacker would need to know the name and location of any executable on the system in order to invoke it successfully.

The vulnerability exists in *Internet Explorer* versions 5.01, 5.5 and 6.0. Although, at the time of the appearance of the worm, *IE 5.01* was the only version that was not vulnerable with the latest available patch, the default installations of *Windows 98SE* and *Windows 2000* come with an unpatched version. As a result, all current out-of-the-box *Windows* systems are vulnerable to this attack.

To exploit the vulnerability, an object has to be created in the script, with codebase pointing to the file stored on the local file system. But how long can that embedded executable be? *Internet Explorer* is very generous in this sense. In my tests, *Windows* PE executables of up to 2 MB could be embedded and executed in a MIME package in the same way as the virus. It would be possible to push higher – but unnecessary, as this size limit is more than sufficient for all contemporary *Windows* malware.

What can be embedded in the package? *Windows* PE executables and old DOS MZ executables can be used (but not COM files, even if they are renamed to have EXE extensions).

The MS01-15 vulnerability has been used by a couple of simple Trojans and in some of the *GFI MailSecurity* test email messages.

But the MS02-15 vulnerability allows the execution of programs only from the local file system, and the worm binary is still in the MIME email. At this point, the MS03-14 vulnerability steps in to help. It allows the referencing of the MIME attachments embedded in the MIME file in the following form:

```
mhtml:{path}\\message.html!File://foo.exe
```

where {path} is the directory from which the code is running, and which is where *Outlook* stores the temporary HTML message. It is a routine task – used in many samples – to extract this working directory, and then the full path can easily be constructed.

The vulnerability regarding the MIME messages resolves this reference to the actual attachment, then passes it as the codebase. As a result, the attachment will be executed. And because the MHTML URL now points to a local file, it will be opened in the Local Security Zone, with higher permissions than in the case of an HTML mail opened in *Outlook Express*.

In fact, when *Windows* resolves the reference, it will save the MIME attachment into the {TEMP} folder. As there is no end boundary in the Mimail sample, anything following the MIME header (namely the binary worm and the dropper script) will be dumped into a temporary file and this will be executed. This temporary file (with the script appended) will be returned by the GetModuleFileName API call. This will be important later.

MHTML URLs can be opened directly from the Run… element of the Start menu. In this case the URL must be 'mhtml:file://c:\message.html!File://foo.exe', just as the worm constructs it. Note that in this case a warning is

displayed, offering to save or execute the program. If execution is selected, another warning is displayed, stating that the application is not properly signed (although, depending on the OS version, this warning may not appear). These warnings are bypassed if foo.exe is accessed from script, as done by the worm (though an error message may be appended to setupapi.log in the Windows folder – but who checks this file regularly?).

This method for dropping and executing a program had been used before the Mimail worm, for dropping two Trojans. The script code was very similar, except that the Trojans in question used BASE64-encoded MIME attachments, instead of the plain binary coding used by this worm. Either method is applicable; the virus writer probably found it simpler to append the script than to implement a BASE64 encoding.

THE BINARY WORM

The original size of the UPX-packed binary was 12,320 bytes, in each generation it is increased by the length of the script dropper (536 bytes). When executed it registers itself as a service process first, thus it will not be visible on the task list in *Windows 9x* systems.

After activation the worm copies itself into the Windows directory as videodrv.exe.

Then it adds the registry value "VideoDriver"="%Windir%\videodrv.exe" to the registry key

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\
CurrentVersion\Run
```

to ensure automatic startup. If the copy operation has been successful (i.e. the virus can open videodrv.exe in the *Windows* folder for write access) then the worm executes that copy and exits, otherwise (if it is already running from the *Windows* folder) it will spread. Instead of a mutex, Mimail uses this method to ensure that only one copy is spreading actively at the same time.

After that Mimail recreates its package in the Windows directory in the file EXE.TMP. It prepends the MIME header to the file returned by the GetModuleFileName API, and then appends the loader script. As the binary component already has the script at the end, the package in each new generation will have another copy of the script.

Next, the worm creates a ZIP package containing EXE.TMP. It will create the necessary ZIP headers for the package, containing the binary in non-compressed, 'store-only' format. This file is also stored in the Windows directory as ZIP.TMP.

With this finished, the worm creates its window named " ", positioned way off the standard display resolution, and the rest of its actions are performed in the *Windows* message handler. If for any reason the window cannot be created, an error message is displayed and the worm exits.

The main spread procedure is executed as a timer callback function, every five seconds. It opens additional threads to accomplish the replication.

The worm calls the gethostbyname API function for the address www.goggle.com to check that the computer is connected to the Internet. If the call fails, the replication routines are skipped for that timer call.

If the call is successful, three new threads are created. In one thread Mimail collects the email addresses from all files except with extensions: '.bmp', '.jpg', '.gif', '.exe', '.dll', '.avi', '.mpg', '.mp3', '.vxd', '.ocx', '.psd', '.tif', '.zip', '.rar', '.pdf', '.cab', '.wav', '.com'. It searches files in the 'C:\Program Files' folder and folders denoted in the registry key:

```
HKCU\Software\Microsoft\Windows\CurrentVersion\
Explorer\Shell Folders
```

This is a long list of folders, which may include (depending on the OS version) the browser cache directory, the My Documents folder of the user, the Favorites, History and similar folders – the best places to look for email addresses.

In another thread the worm sends out the infected messages. The collected addresses are stored in the file eml.tmp in the Windows directory. During the replication the addresses are read back to memory from this file. For each email address the worm performs a Mail Exchange lookup for the target domain using the DNS servers defined for the current host. If the MX can be located Mimail connects to it and delivers the mail using SMTP.

If a DNS server is not found, it will default to 212.5.86.163 (the name server of a Russian fashion store, lemonti.ru – probably related to the virus author, who might be suspected of being of Russian origin). The worm thus directly contacts the mail server assigned to the destination address. This way if someone cares to check out the email headers of an infected message, it would still seem as if it is coming from the target domain – the signs of spoofing in this case are not as obvious as using the usual technique of sending via the local SMTP server.

The infected messages appear to come from the admin@ account on the local domain (the virus spoofs the sender address). The messages contain the '*X-Mailer: The Bat! (v1.61)*' line in their message header to make it look as if it has been mailed from an email client. The subject line is the following:

```
your account          {text}
```

where {text} is a randomly generated string built using lower-case characters of the English alphabet.

The body of the message is as follows:

```
Hello there,
I would like to inform you about important
information regarding your email address. This
email address will be expiring.
Please read attachment for details.
—
Best regards, Administrator
{text}
```

Here {text} is the same as in the subject.

It is most likely that the random characters are intended to ensure that the outgoing messages have different subject lines, in order to bypass spam filters, as spammers often use this technique.

The virus is attached to the outgoing messages. The attachment name is message.zip. The content of this ZIP file is message.htm, which contains the worm and a short Javascript code that drops and executes it.

The worm also captures text from the foreground windows and sends this data to specific email addresses.

GENERATION TIME ANALYSIS

From time to time I give presentations about email viruses, in which I compare their spread with traditional, usually boot, viruses. The rapid spread of email viruses is attributed to their large multiplication ratio (they send themselves to several addresses in a single propagation act) and their short cycle time (the time elapsed between two replication cycles). The cycle time depends roughly on how often an average user reads their mail (and executes the attachment, whether deliberately or accidentally), and on how fast an email travels from source to destination (usually only a matter of minutes).

Until now I had only subjective estimates of the latter figure, placing it between 30 minutes and an hour. However, Mimail gave us a real-life 'measurement' of this cycle time. As a result of a bug in the replication routine, the worm appends its script to itself each time it replicates. Therefore, simply by counting the occurrences of this script in the infected message we get its generation number.

Using data collected by *MessageLabs* (thanks to the generous help of Alex Shipp, who provided me with first occurrence data for each generation), it is possible to determine how long it took for the virus to jump from one generation to the next.

The table shown below summarizes the results from generation 1 to generation 25, listing the relative appearance of each generation from the first, in hour:minute format.

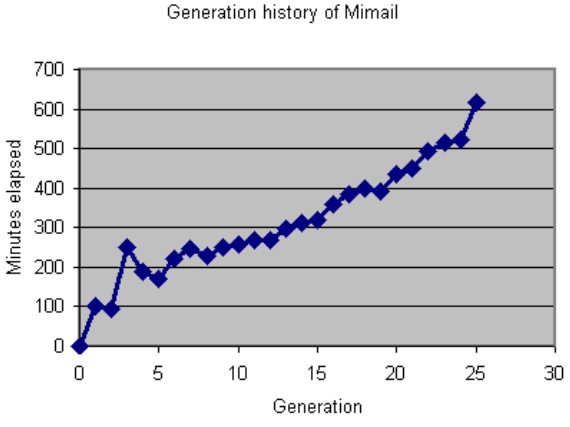
Generation	0	1	2	3	4	5	6	7	8	9	10
Rel. time from start	0:00	1:40	1:35	4:12	3:10	2:52	3:40	4:07	3:47	4:09	4:18

Generation	11	12	13	14	15	16	17	18	19	20	21
Time	4:29	4:27	4:59	5:12	5:19	5:59	6:23	6:40	6:30	7:17	7:31

Generation	22	23	24	25
Time	8:15	8:36	8:42	10:18

Using this data some nice charts can be drawn, as illustrated opposite.

Although we do have data up to generation 33, the part missing from this table cannot be used for evaluation purposes. The reason is simply because the virus appeared



on a Friday afternoon, and the later generations appeared over the weekend, during which time 'normal' users (i.e. the targets of the virus) were not spreading the virus. During this period larger time lags were observable – more than a day elapsed between generations 30 and 31. Also, the data for the first five generations is strongly biased as a result of the low number of viruses in circulation in these generations.

Generations 5 through 25 can give us an excellent estimate for the cycle time of the worm – which, using some regression analysis proved to be 18.3 minutes (more precisely, the cycle time is between 16.2 and 20.4 minutes, with 95% confidence level).

It turned out that my original estimate (30–60 minutes) for the cycle time of a typical email worm was an overestimate, and in reality it takes only about 18 minutes for a worm to make a full infection circle. Because of the ZIP packing this virus required additional clicks for activation (compared with many other worms), so this cycle time could be even lower for an average worm, but not too much.

CONCLUSION

Some worms make use of *Windows* security holes to execute attachments without the need for the user to double-click. Other worms, like Mimail, use social engineering to achieve the same goal. In this case the users were made to believe that the message came from the sysadmin of their domain.

Even though the virus was packed in a ZIP archive, users carried out the extra two mouse clicks (either using applications like WinZIP or the native ZIP support of *Windows XP*) that were needed to activate the attachment. Mimail easily bypassed Klez.H on the top of virus prevalence charts, proving that social engineering can sometimes be more effective than automatic activation.

VIRUS ANALYSIS 2

YOUR MUMU.B DON'T DANCE...

Rodelio Fiñones, Jaime Lyndon 'Jamz' A. Yaneza
TrendLabs, Trend Micro Inc., Philippines

In June 2003, we were alerted by a number of customer inquiries regarding the release of a new variant of the MUMU worm.

Like its other variants, this version of the worm spreads through the *Windows NT*, *2000*, and *XP*-based system default administrative shares. It propagates by scanning the whole Class C network and connects to systems with weak administrator accounts and passwords using a dictionary attack. The first variant of the worm, BAT_SPYBOT.A, used a collection of batch files and hacking utilities to infiltrate vulnerable systems, while this variant embeds all its components in a single Win32 executable and extracts them along the way. The later variant also contains new features such as key logging and process termination.

AND IT'S TIME TO GO TO TOWN...

Upon execution, the worm creates two mutexes, named 'qjinfo1mutex' and 'qjinfo2mutex'. This is to ensure that only one instance of the worm is running on the system.

While active in memory, the worm parses its resources to extract the following components:

- %System%\last.exe (20,480 bytes)
- %System%\kavfind.exe (30,208 bytes)
- %System%\IPCPass.txt (510 bytes)
- %System%\psexec.exe (36,352 bytes)
- %System%\mumu.exe (294,912 bytes)

The file LAST.EXE is executed immediately as a child process. It is responsible for the key logging and process termination routines. KAVFIND.EXE (which is detected by *Trend Micro* as TROJ_HACLINE.A) is a UPX-compressed and multi-threaded hacking tool that can be used for scanning IPC shares on remote machines. It uses the file IPCPass.txt as an optional input file to contain the list of usernames and passwords. *Trend* detects the file IPCPass.txt as BAT_SPYBOT.A.

PSEXEC.EXE is a legitimate remote process tool produced by *SysInternals*, albeit compressed by UPX. Finally, MUMU.EXE is a copy of the original worm that is distributed on the vulnerable systems in the LAN.

When the file, LAST.EXE, is active in memory, it decompresses and creates another mutex named 'qjaashyuhv1.0'. Next it extracts the embedded components %System%\bboy.dll (36,864 bytes) and %Windows%\bboy.exe

(20,480 bytes), as well as creating the following auto-run registry entry:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\
CurrentVersion\Run]
Kernel =”%Windows%\bboy.exe”
```

This ensures that this worm component is executed on system startup. Since most of the worm functionalities are performed by components, the main worm program does not need to be registered in any auto-run keys.

Furthermore, this component remains resident in memory until the system has been rebooted or until a user terminates it manually. While in memory, it drops the file BBOY.EXE continuously and creates the above-mentioned registry entry.

Under *Windows 9x* systems, the worm registers itself as a service process such that its process name is not visible in the Windows TaskList (which is visible by pressing CTRL-ALT-DEL).

WHEN EVENIN' ROLLS AROUND…

The keylogging and process termination process begins by loading BBOY.DLL in the address space of the current process using the LoadLibrary API. Then it retrieves the entry-point of the BBOY.DLL exported function InstallHook and invokes it at once. The InstallHook API initiates monitoring by invoking the SetWindowsHookEx *Windows* API, which then installs an application-defined hook procedure in the hook chain. The worm specifically monitors messages that are posted in the *Windows* message queue – those that are associated to all the threads running in the same desktop as the calling thread (in this case LAST.EXE/BBOY.EXE thread). Subsequent calls to GetMessage and PeekMessage APIs will allow the worm to retrieve the information from these messages. All recorded information is stored in the file QJINFO.INI.

Another thread reads the contents of the log file continuously. Prior to sending the log file, this worm thread checks for the existence of the process MAIN (the extension is ignored). Note that this file is not included as one of the worm's components. It is possible that this file belongs to other malicious worms or possibly is one of the components for the next variant of the worm. It will send the log file only if this process is running on the system. Using a separate thread, it sends the file to the email address specified at fix offset 40H of the current program (it could be LAST.EXE or BBOY.EXE) via connection to the web mail site www.58589.com. The email has the following characteristics:

```
From: babyj@8848.com
To: terminal2000@163.com
BCC: cq@58589.com
```

The worm also attempts to terminate the following processes:

kvapfw.exe

kvfw.exe

DFVSNET.EXE

PasswordGuard.exe

EGhost.exe

Iparmor.exe

pfw.exe

JUST ABOUT TO MOVE IN…

In order to replicate in the network, the worm utilizes a three-year-old 'NT LsaQueryInformationPolicy() Domain SID Leak Vulnerability' (*CVE 2000-1200*) to retrieve *NT* domain's SID from workstations within that domain. The SID can then be used to obtain a list of usernames. Showing similarities with the previous variant of this worm, it also uses a weak username and password dictionary attack to penetrate systems. It uses the hacking tool KAVFIND.EXE with the input file IPCPass.txt to connect to port 139 (SMB over TCP/IP) of the remote machine within a Class C network. The following is the list of usernames and passwords listed in the file IPCPass.txt:

%null%	88888888	public
%username%	5201314	private
%username%12	pass	default
%username%123	passwd	1234qwer
%username%1234	password	123qwe
123	sql	abcd
1234	database	abc123
12345	admin	123abc
123456	root	abc
1234567	secret	123asd
12345678	oracle	asdf
654321	sybase	asdfgh
54321	test	!@#$
1	server	!@#$%
111	computer	!@#$%^
11111	Internet	!@#$%^&
111111	super	!@#$%^&*
11111111	user	!@#$%^&*(
000000	manager	!@#$%^&*()

00000000 security KKKKKKK

888888

After two minutes of execution, the worm inspects the output file IPCFind.txt for the list of remote machines that can be exploited. The format of the output is

```
<IP Address>  <user name>/<password>
```

It connects to each exploitable IP address using the command

```
net use \\%ip_address%\admin$ "%password%" /
u:"%user_name%"
```

where

`%ip_address%` = the IP address of the target machine

`%password%` = the password obtained from IPCFind.txt

`%user_name%` = the username obtained from IPCFind.txt.

Note: NET is a legitimate and default-installed *Windows* application.

Once the connection is established, the worm MUMU.EXE is copied to the \\%ip_address%\admin$\system32 folder and PSEXEC.EXE is used to execute the worm remotely. The worm uses the command below to accomplish this task:

```
start /i /min /wait /B psexec \\%ip_address% -
u "%username%" -p "%password%" -d MUMU.EXE
```

Aside from IP scanning, it also attempts to distribute the worm to all the active TCP connections. It executes the following command to retrieve the list of connections and saves it to a temporary file:

```
CMD /c netstat -n|find ":" >A.TMP
```

After one minute of execution, it parses the temporary file A.TMP to ping every IP address where connection state was set to SYN_SENT. The connection is assumed to be active once a 'reply' string is received. In that case, the worm connects and copies MUMU.EXE to the admin$\system32\MUMU.EXE. Then the worm file is executed remotely using PSEXEC.EXE tool. The network ID of the network is also stored in this registry path:

```
HKEY_LOCAL_MACHINE\SOFTWARE\mumu
```

THE OLD FOLKS SAY..

The worm also sends the log file QJINFO.INI via SMTP to a server on *smtp.sina.com.cn*. It retrieves the mail service using GetServByName API and generates an email with the following characteristics:

```
From: reint0.student@sina.com
To: sendmail2.student@sina.com
Subject <X><computer name>
Date: <current_date>
```

Note: <X> is a one-byte random number.

The worm also creates a hidden pop-up window with the class name 'Systary' and adds this registry entry:

```
HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\
CurrentVersion\Run
       Folder Service = qjinfo.exe
```

Note: QJINFO.EXE is not included in the worm components. It is possible that other currently unknown or forthcoming variants of MUMU will use this file.

CONCLUSION

WORM_MUMU.B proves another point that network-aware worms, coupled with exploit code, can successfully wreak havoc in a company's network. We are seeing a trend in which standard *Windows* administrative tools, commonly used by network administrators for network maintenance, are being used maliciously against them.

The issue of detection depending on how such tools are used is now in the minds of users and poses another daunting task for the anti-virus industry, given the limited ways in which the nature of these tools could be ascertained. Should this now be part of the functionality of an anti-virus product?

The overall success of this worm is a result mainly of what we call the 'human element' – or rather the way in which 'social engineering' comes into play. The need to improve user education and awareness is still of paramount importance in this situation.

WORM_MUMU.B

Type:	Network worm.
Aliases:	W32.Mumu.B.Worm, W32/Mumu.b.worm, Win32/HLLW.Mumu.A, Worm.Win32.Muma.c, W32/Muma.B, Win32.HLLW.Mumu, Worm/Mumu.B.1, Worm/Muma, Worm.Win32.Muma.294912.
File size:	294,912 bytes.
Payload:	Performs multi-threaded IP scanning, process termination, key logging.
Removal:	Terminate worm processes. Delete malicious files. Remove registry entries added by the worm.

VIRUS ANALYSIS 3

BLAST OFF!

Peter Ferrie, Frédéric Perriot, Péter Ször
Symantec Security Response, USA

On 11 August 2003 – the same day it was completed – a 6176-byte-long UPX-compressed bug started to invade the world using a recent vulnerability described in *Microsoft*'s *MS03-26* security bulletin. Even *Windows Server 2003* was affected by this vulnerability. Patches were made available by *Microsoft*, but on this occasion there was only a short delay between the announcement of the vulnerability and the appearance of the worm that exploited it.

Users of *Windows XP* had a chance to get the patch applied automatically via Windows Automatic Updates. However, the same cannot be said for the *Windows 2000* platforms, where users would need to pay closer attention to the update procedures.

ALL SYSTEMS GO

The first thing Win32/Blaster does when it runs on a system is to create a value 'windows auto update' in the 'HKLM/…/Run' registry key, pointing to the bare file name 'msblast.exe' (for variant .A). This relies on the assumption that the executable has ended up in a directory that *Windows* searches by default, which is usually the case.

Then the worm attempts to create a mutex named 'BILLY', and aborts if the mutex exists already, in order to avoid multiple instances of the worm running at the same time.

Win32/Blaster then waits for an active network connection, and starts searching for machines to infect.

SP4, SP3, SP2, SP1, IGNITION!

The target selection in Blaster is somewhat different from that found in CodeRed and Slammer. Sixty per cent of the time, Blaster will go after entirely random IP addresses, and the other 40 per cent of the time it will attack machines on the same class-B-sized network as the host, hoping to take over pools of vulnerable systems on the local area network.

The scanning for targets is linear (the target address is increased monotonically until it reaches the end of the IP space) and, in the case of a local attack, starts at or slightly below the class-C of the host.

The worm targets machines running *Windows 2000* and *Windows XP*, and intentionally favours the exploitation of *Windows XP* machines (probably because the payload relies on the increased availability of raw sockets there – the requirement to be administrator was removed). Eighty per cent of the time, the exploit is tuned for *Windows XP* systems, the other 20 per cent for *Windows 2000* systems. This selection is made only once, whenever the worm initializes.

All unpatched Service Packs of both *Windows XP* and *Windows* 2000 systems are affected, but because of this random tuning, the worm will sometimes just cause a denial of service (DoS) on the attacked machines, crashing the RPC service.

SECOND STAGE – THE SHELL

The infection of a new machine is a three-phase process, involving quite a lot of network activity in comparison with the single-connection CodeRed and the lightweight Slammer.

First, the worm sends its attack buffer over port 135/tcp, which exploits the RPC DCOM vulnerability and causes the remote machine to bind a shell in the SYSTEM context ('cmd.exe') to port 4444/tcp.

Second, the worm sends a command to the newly created shell to request a download of the worm file from the attacking host to the victim. The transfer is done over port 69/udp using the tftp protocol (the worm implements its own crude tftp server which formats sent data according to RFC 1350, and uses the tftp client that is present by default on most *Windows* systems).

Finally, once msblast.exe has been downloaded successfully, or after 21 seconds, the worm requests the remote system to execute the downloaded file.

HOUSTON, WE HAVE A PROBLEM

Once the shell exits, the hijacked RPC service thread running the shell code calls ExitProcess(), causing the service to terminate. The termination of the RPC service, regardless of how it occurs, triggers a reboot in *Windows XP* systems after one minute. On *Windows 2000* systems, the termination will result in a variety of unusual side effects, among the most critical of which is the inability to use the Windows Update web service.

PAN GALACTIC GARGLE BLASTER

As is common for fast-spreading worms, Win32/Blaster reuses an exploit code that was posted previously to various security mailing lists. The exploit uses two so-called 'universal offsets' as return addresses in a classic stack buffer overflow, each of which is compatible with multiple service packs of one *Windows* version. The vulnerability is located in the code of the rpcss.dll file, in a function related

to the activation of DCOM objects. The buggy function extracts a NetBIOS server name from a UNC path specified by a DCOM client, and attempts to place it into a 32-byte buffer on the stack, without bounds checking.

Once the stack is smashed, the hijacked return address leads to a 'call ebx' instruction (in a 'well-known' constant data table) which then jumps back to a nop ramp in the shell code. This is possible because the ebx register is pointing to a local variable in an earlier stack frame (i.e. at a higher memory address) created by the fourth-level (!) caller of the buggy function (see Figure 1).

The shell code retrieves some useful API addresses, binds to port 4444/tcp, accepts one incoming connection, spawns the shell and ties its input to the port 4444 socket, waits for the shell process to finish, then exits.

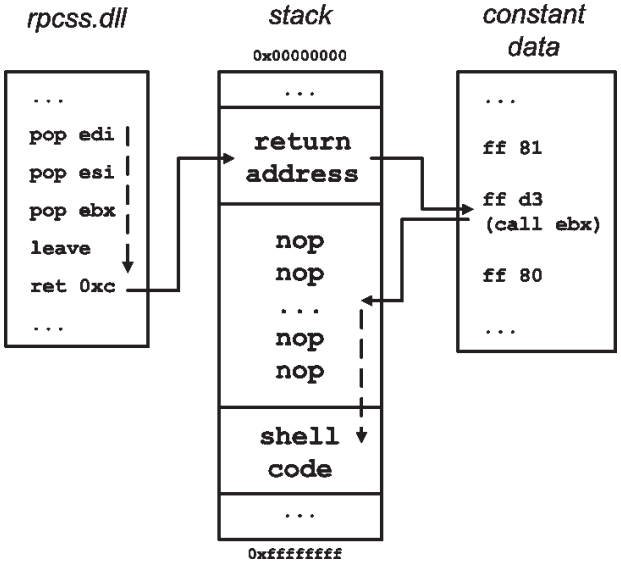


Figure 1: Memory layout and control flow.

MS-DoS

Win32/Blaster implements a SYN-flooding Distributed Denial of Service attack against the website 'windowsupdate.com', an alias of the main *Microsoft* Windows Update site.

The attack is carried out if the day of the month is greater than 15, or the month is greater than 8, i.e. every day from 16 August to the end of the year, and then starting again on 16 January until 31 January, 16 February until 28(/29) February, and so on.

To generate the traffic, the worm uses raw sockets. DoS packets have spoofed source IP addresses, the two low octets are randomized for each sent packet; the high octets are either taken from an IP of the source host, or otherwise initialized once to random values.

The traffic features various characteristics that can help in recognizing it: the two low bytes of the TCP sequence number are always zero, the source port is between 1000 and 1999 (inclusive), and the IP identification field always has a value of 256.

CONCLUSION

The first use of a command shell attack by a Win32 worm has finally arrived. The spawning of a shell had previously been used only by Win32 exploits and by Unix worms executing /bin/sh with system calls such as system() or execve().

Windows worm writers are slowly merging existing exploit code with their creations to make them more harmful. The tendency that started with CodeRed, Slammer and other Unix worms, continues. The delay between the appearances of such creations seems to have decreased from a year to six months.

It is evident that among defensive technologies, proactive behaviour blocking techniques will become essential to fight back against such 'cloned' worms in the future. Peter Ször's paper, 'Attacks of the worm clones – can we prevent them?' (to be presented at this year's RSA Europe conference in November) uncovers the details of how we can get closer to this goal and demonstrates research prototypes that work effectively against this clone.

This time not only corporate servers risked being affected; the threat had the potential to reach the majority of *Windows* desktops. This is 'Buffer Overflow for the Masses'.

Win32/Blaster

Size:	6176 bytes.
Aliases:	W32.Blaster.Worm, W32/Lovsan.worm, Win32.Poza.A.
Type:	Worm, exploits buffer overflow vulnerability in DCOM/RPC (described in Microsoft's MS03-026 Security Bulletin).
Trigger:	DoS attack attempt against windowsupdate.com after 16th of each month or after August each year.

FEATURE 1

DIGITAL SIGN: THE NEXT TARGET?

Dr Ferenc Leitold
Veszprém University, Hungary

The tram drivers in Budapest often warn the passengers: 'Beware! There are pickpockets on the tram – look after your property!' Public messages like this one have two effects: the passengers hold their bags tighter, and any potential pickpockets are alerted to a good opportunity that could be exploited now or in the future. This article attempts to present the possible points of attack relating to the use of the digital signature. Of course, this warning may also have two effects. Nevertheless, it is in the interest of everyone using or accepting digital signatures to be aware of the dangers when they are using the system.

The security problems which are being made public day by day and the continual appearance of new types of virus are undermining the security of the electronic signature. In the hardware and software (including operating systems) environment there is potential for the digital signature to become the target of virus attacks. A computer virus or worm can take control of the victim's computer, where it can create a new signature or, by manipulating the signing process, counterfeit an approved signature.

A further problem is the forwarding of confidential documents or messages. For this PKI tools provide an excellent solution. But the forwarding of an encoded message – especially through a firewall with virus protection – raises several security issues.

DIGITAL SIGNATURE

The algorithm of the digital signature uses the algorithm of the public key encoding (e.g. RSA). The digital signature works as follows. Using the binary code series of the document to be signed, a fingerprint peculiar to that document is prepared. This process can be carried out with the help of the Hash algorithms. The fingerprint is then ciphered with the cipher part of the public key algorithm. The code series prepared this way is the *digital signature* rendered to the document. Afterwards the sender forwards the document together with the digital signature rendered to it. The recipient receives the document and the digital signature and, with the help of the same Hash algorithm, he prepares the fingerprint rendered to the document. Using the sender's public key, he also prepares the fingerprint rendered to the digital signature. If the two fingerprints are identical, he can make sure that the digital signature was made with the cipher pair of the public key used for the supervision. The mathematical theory of the method does NOT ensure that the digital signature has been rendered to the person signing the document or that the digital signature has been made with the knowledge of the owner of the cipher key.

TRADITIONAL SIGNATURE – ELECTRONIC SIGNATURE

When we sign a document on paper we rely on our eyes and our mental ability to make sense of what we see. Our eyes will give evidence to the fact that the signature is put only onto the document that we intend to sign.

When we prepare an electronic signature we must believe that the information displayed on the screen corresponds to a bit series stored in the memory or the mass storage. We must believe that the unit constructing the signature (e.g. an external card reader connected on a serial port or USB) provides only that bit series with the electronic signature whose correspondence is displayed on the screen.

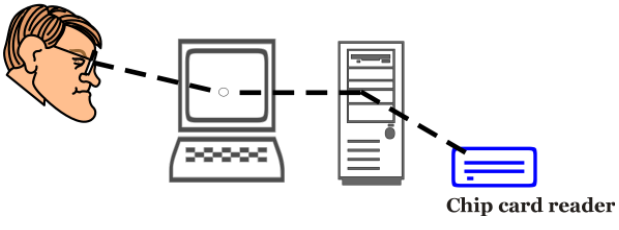


Figure 1. Electronic signature.

When we use a multi-purpose computer to prepare electronic signatures then we must completely trust its hardware and software installation and the proper operation of the software. Of course, we cannot check this in any visible objective way. Thus, there are two opportunities for a potential attacker:

- to affect the presentation
- to manipulate the signing procedure.

AFFECTING THE PRESENTATION

We ought to expect the document that is to be signed to contain all the information required to interpret and present it. If information from another source is required then the image presented of the document may be affected. For example, *Word* and certain PDF documents do not contain

all the fonts required to present the image of a document. Thus, in a different environment from that in which the document was created the fonts may be substituted and the image of the document will be different in the new environment. Unfortunately, ASCII text files are no different. Despite the fact that there are no fonts here we must know the image of the characters for the presentation. This is information outside the document (the bit series of the text), which is fixed by the ASCII standard, but the presentations are made by the hardware and software of the computers. In the case of a VGA card the image of the characters can be overwritten! The problem is unrelated to operating system. The notion of fonts exists under *Linux* and UNIX systems too and a *StarOffice* document does not contain the images of the letters either. To ensure the image presented is an accurate representation of the document it is vital that the document itself contains the binary images of the characters.

What opportunities are available to an attacker to exploit the gap in the security system?

1. If the attacker has access to another computer he can change the image of the letters in any of the fonts. Applications that do this are freely available on the Internet (see Figure 2).
2. An attacker may create his own program to change the letters.
3. The attacker can send this program via email. A tremendous number of viruses are sent by email today.

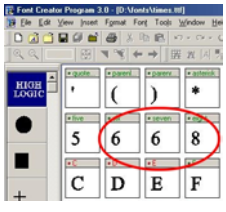


Figure 2. Easy font modification.

A malicious attacker could easily load a program onto a victim's computer, which ensures that the signer will see something different from what he intends to sign. He can also eradicate himself entirely from the victim's computer following the signing, e.g. at a definite time. After that the user would try to prove his honesty in vain; the electronic signature is an approved evidence in the courts of some countries.

MANIPULATING THE SIGNING PROCEDURE

If we are fully aware of what we intend to sign – or at least we believe that we are – we can provide the document with our electronic signature. In order to do this we need a signature-making device.

The signature-making device contains software as well as hardware elements, and ensures that all conditions are fulfilled in order to carry out the signing procedure without any further interaction. If we use some kind of a chip-card, after inserting the card every condition is given for the signature. We cannot check manually whether we render our signature to that particular bit series and we cannot make sure that there is no signature rendered to other bit series.

The following method could be used for an attack: a malicious program, which is loaded onto the computer in one way or another watches the interactive activity the user must carry out after satisfying every condition for the signature (e.g., he has entered the chip-card into the reader). The malicious program senses what information the user program sends to sign for the card reader. The malicious program sends this information to the reader and waits for the response. It does not forward it to the user program but sends another bit series for the reader to sign. When this has happened, the malicious program sends back the signed answer to the user program. All this happens so quickly that the user does not notice anything. Like the majority of email viruses the malicious program may even send the signed bit series back to the attacker using its own SMTP routine.

USING DOCUMENTS WITH A DIGITAL SIGNATURE

The advantage of the digital signature is that the two signatories of a common declaration (e.g. a contract) need not meet in person. It is sufficient to exchange the electronically signed declarations. Nevertheless, everybody prefers to handle their contracts discreetly and would not like any unauthorised party to have access to them. PKI offers an excellent opportunity to avoid such unauthorised access by ciphering the messages.

The electronically signed document must be sent to the receiving party. We can do this through data media or by mail. Either way the solution is not less comfortable than in the case of the traditionally signed paper document. The only significant difference is that when forwarding through data media we can send or carry the information in ciphered format.

A natural way of forwarding a document is sending it through the Internet. Sending a message on the Internet is about as safe as sending information on a postcard – pretty much anyone can read it – therefore it is essential to cipher the document.

Today, every business or institute has an internal information infrastructure or inner network. It is very

common for there to be a firewall in place at the meeting point of the internal network and the Internet. The firewall watches the traffic between the inner network and the Internet and tries to protect the internal network from the dangers coming from the Internet. A well-configured firewall system must have packet-filtering devices and content-filtering abilities (e.g. virus protection) too.

Let us assume that two managers intend to exchange their electronically written documents on the Internet. The easiest way to do this is to send the signed message through email. It is essential for both of them to keep the content of the document secret, even within the internal network. Therefore they cipher their messages, which can be done easily with PKI technology.

The real security gap occurs at the firewall. The system managers maintaining the firewalls have two options:

1. They configure the firewall so that it does not allow documents through if their contents cannot be checked. But, in this way, ciphered and signed documents will never get through to the other party.
2. The firewall is set to let through ciphered messages without supervision. Then the two managers can exchange the signed and ciphered messages. But this lapse in security is sufficient for an attacker to send a malicious program through the firewall – if it is ciphered with the manager's public key (see Figure 3).

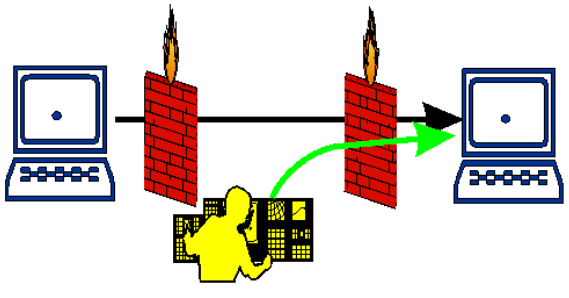


Figure 3. Opportunity to attack through the firewall.

SUGGESTIONS

Being aware of the security problems described above we cannot claim that the use of the digital signature is perfectly safe. If we assume that the digital signature is made on a computer that is used for multiple purposes, we must face serious security problems.

It does make a difference what we sign, or rather, what we sign can be interpreted only in the way we mean it. It does make a difference what system and what device we are using for the signature. And, finally, it does make a difference for what purpose we intend to use the signed document and how we intend to forward it.

One of the shortcomings of the legal regulations in many countries is that they do not specify unambiguously the types of data that can be signed electronically. The legislation ought to define 'text' and 'letter'. Which are the electronic forms that can be regarded as 'text communicated with letters'? Is a scanned A4 page saved in a binary picture format acceptable if it contains letters only?

It is also a basic expectation that the regulated forms and standards should be made public and accessible for all, otherwise how could we supervise a document based on a ciphered form? The supervision could be assisted with supervising software with an open source code.

Regulations should be brought into effect regarding the forwarding of documents that have been signed digitally. At the moment a few of the regulations about the digital signature excludes the use of the keys for other (i.e. ciphering) purposes. The ciphering keys could be classified similarly to the keys used for the signatures. With a common regulation the providers of the authentication could carry out both authentications a lot more easily than doing so separately.

I believe users should be made aware of the potential sources of danger. This could be done by the providers of authentication because they have knowledge of the devices used for digital signatures, and ought to provide a set of guidelines for their secure use.

Users – whether private individuals, business enterprises or public institutions – are interested in the secure operation of their systems. The security of the signature-making devices is closely related to the overall security of the computer. Making the computer more secure by installing a firewall and/or virus protection or a set of security guidelines will make the process of creating the digital signature more secure too.

There is no solution to the security of the digital signature that can be detached from the overall security of the computer.

CONCLUSION

The rendering of an electronic signature to a document raises a number of security problems when we use a computer for making the signatures. The reason is that there is no operating system (probably there cannot be any) which could provide sufficient security for making digital signatures at the moment. Users must maintain an overall security culture, which can help to prevent the potential problems.

FEATURE 2

CHALLENGES IN GETTING 'FORMAL' WITH VIRUSES

Arun Lakhotia and Prabhat K. Singh
University of Louisiana at Lafayette, USA

Is it a virus, a worm, a Trojan, or a backdoor? Answering this question *correctly* for any *arbitrary* program is known to be an undecidable problem. That is, it is impossible to write a computer program that will identify correctly whether an arbitrary program is a virus, a worm, etc. – no matter how much computing power is thrown at the problem.

With the emergence of polymorphic and metamorphic viruses, the anti-virus community is finally beginning to face this theoretical limit. Signature-based heuristics, whether dynamic or static, for detecting malicious code are no match for a program that modifies, encrypts and decrypts its code as it propagates.

Researchers in academia and industry are beginning to develop anti-virus technologies founded on formal methods of analysing programs (Christodorescu and Jha 2003, 12th Usenix Security Symposium, 2003; Perriot, 13th Virus Bulletin International Conference 2003; Singh, Moinuddin et al., 2nd European Conference on Information Warfare and Security, 2003). These methods, with rigorous mathematical foundation, have mostly been developed for optimizing compilers and, more recently, for hardware and software verification.

The mathematical guarantees offered by these methods are a necessity for compilers and verifiers, for it is unimaginable that one would use a compiler or a verifier based on heuristics that gives correct results only 90% of the time. The success of these techniques in compilers and verifiers has been extrapolated to offer promise in anti-virus technologies.

We argue that the formal methods for analysing programs for compilers and verifiers when applied in anti-virus technologies are likely to produce good results for the current generation of malicious code. However, this success will be very short-lived for it is extremely easy to 'attack' these analysers to make them produce incorrect results.

The fundamental basis of our observation is that the formal methods designed for optimizing compilers assume that the compiler and the programmer are allies. In other words, the programmer does not stand to benefit from breaking the formal analysis. On the contrary, the compiler writer can assume that a knowledgeable programmer may be willing to reorganize a program to get optimal results from the compiler.

If there is one thing we can learn from polymorphic and metamorphic viruses it is that virus writers enjoy the challenge of beating the anti-virus technologies. Also, it would not be inaccurate to say that the virus writers, or say, the virus-engine writers are not just good programmers; they have a very good understanding of computer science. It is by no means a small feat to write a program that disassembles a host program; reorganizes the code; injects malicious code deep inside this reorganized code; reassembles the new program; and overwrites its disk image in the correct format. And this has not even touched the capabilities needed to encrypt, decrypt and morph the malicious code.

It should, then, be safe to assume that virus writers can and will find ways to break the formal analysis techniques as well.

Are we saying that it is not worth using formal methods in anti-virus technologies? Not really. Just as signature-based heuristics (however limited) have offered effective defences by raising the bar for the virus writer, so will the use of formal methods. However, these methods must be adapted to the new scenario where the programmer (i.e. the virus writer) and the analyser have conflicting goals.

Rather than acting like an ostrich, it is important that we assess the use of these methods and their effectiveness against attack on the analysis mechanism itself. This is not an easy challenge, for as we show, it calls for a complete rethinking of all the underlying assumptions in all phases of analysing a program.

In this article we enumerate, using an example, the promises and pitfalls of formal analysis. We then outline the steps traditionally used in performing such analyses. Next we show how the known algorithms for each step can be broken. We conclude the article with a call to rethink the process of analysing binary executables.

PROMISES OF FORMAL ANALYSIS

The methods for formal analysis of computer programs have mostly been motivated by one of the following: (1) improve runtime performance, (2) decrease code size and (3) increase confidence in the correctness of a program – all with minimal, if any, intervention from the programmer.

The formal analyses may be classified into two categories: static analyses and dynamic analyses. A static analysis finds properties that hold for *all* executions of a program. Such an analysis is typically performed without executing the program, hence the qualifier *static*. A dynamic analysis finds properties true for a specific input. It is called *dynamic* because it is typically performed by executing or interpreting the program.

The classic compiler optimizations, such as dead code elimination, constant folding, constant propagation, elimination of partial redundancies, loop unrolling, etc., are static analyses. Model checking, a technique for verifying the existence or non-existence of certain temporal properties in a program is also a static analysis. Analysing coverage achieved by a given set of test cases is a dynamic analysis. Profiling a program to identify code segments or functions consuming the most time is also a dynamic analysis.

We now take a common analysis performed by an optimizing compiler and assess how it may be applicable in anti-virus technologies. Consider the following code segment:

```
int offset, port;
offset = 2.5*2;
port = offset + 20;
```

Instead of generating code to multiply and then add the constants, as in the above code segment, most optimizing compilers will use constant folding to generate code equivalent to the following:

```
port = 25;
```

Quite coincidentally, metamorphic viruses apply transformations that go the other way around. They replace a constant by a sequence of steps that eventually produce the same constant. We call such a transformation *constant unfolding*. A metamorphic virus may apply constant unfolding randomly to different constants appearing in the program, thereby generating code that looks different from the original. Besides changing the signature of the program, this transformation also obfuscates the program, making its manual analysis harder.

Constant folding and constant unfolding are inverses. It stands to reason that an anti-virus technology may use constant folding to undo the obfuscation created by constant unfolding. For instance, in order to determine whether a program may be sending email, an anti-virus system may check whether a program calls the *connect* function so as to connect to port 25. If a virus writer, or a metamorphic engine, attempts to obfuscate the computation of the port number, the anti-virus system may apply constant folding to de-obfuscate it.

Like constant folding, other optimization transformations, offer similar promises. Dead code introduced by a metamorphic virus could be removed using dead-code elimination; reordering of instructions by introducing jump statements can be undone by reordering the instructions.

Though, at first glance, optimizing away the effects of metamorphic transformations looks like a promising technique, the real test of such analysis techniques depends on how they can be attacked. The example used above can be optimized by constant folding because the computation generating the constant is in the same control-flow block. Consider the following code:

```
if (x < y) {
    x = 10;
    y = 15;
} else {
    x = 15;
    y = 10;
}
port = x + y;
```

In this program too the variable ‘port’ has the value 25 for all possible executions. However, constant folding will be unable to determine this fact. The reason is that ‘port’ depends on the value of variables ‘x’ and ‘y’, and neither of these variables holds a constant value at the point at which ‘x+y’ is computed. Hence, the analysis will incorrectly determine that ‘port’ is not a constant.

This limitation of static analysis should not come as a surprise. Determining whether a piece of code always produces a certain constant value is the same as determining program equivalence, which is an undecidable problem. Hence, we can never develop a method that always determines correctly that a variable is a certain constant for all possible programs in which the variable is really a constant.

One may argue that dynamic analysis may be used to make the determination needed in the above example. But then, dynamic analyses can also be fooled. For example, to avoid getting into an infinite loop, and thereby hanging a system, an anti-virus system that uses an emulator (or a sand box) would have to determine when to terminate the analysis. The virus could attack the analysis by exhausting the patience of such an analyser.

PROCESS OF ANALYSING BINARIES

The real challenge facing the anti-virus community is in analysing binary viruses. The anti-virus community knows how to handle macro viruses pretty well. In this section we outline the steps commonly used in analysing binaries. In the next section we highlight some of the assumptions at each step and how they can be attacked.

Figure 1 gives a diagrammatic representation of the steps in analysing a binary statically for the presence of malicious behaviour. For the sake of our discussion we assume that the input binary is unencrypted. This constraint disqualifies the use of this analysis for polymorphic viruses. Even without polymorphism we have enough to worry about, hence we ignore this dimension.

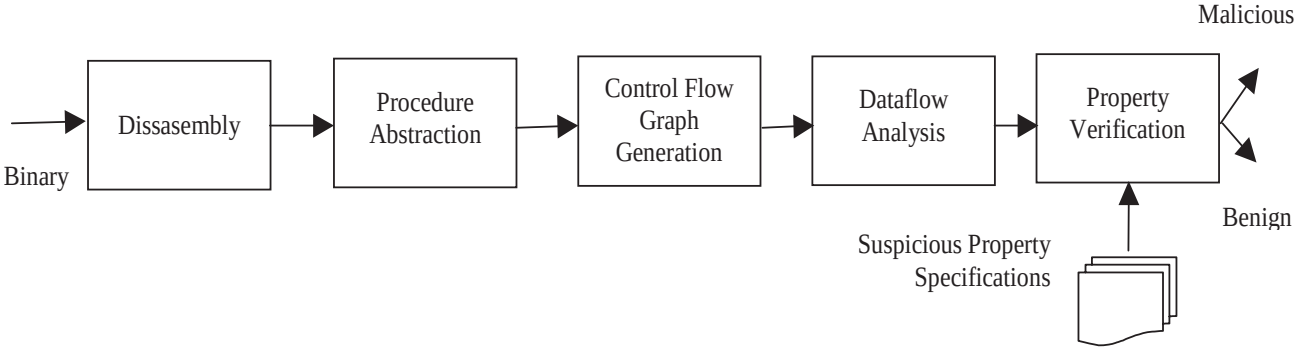


Figure 1. Stages in static analysis of binaries.

The various stages in the analysis and the key function performed at the stage is described as follows:

Disassembly

The broad role of disassembly is to create mnemonic representation of binary code. The mnemonic representation is useful for manual analysis. However, the mnemonics are not necessary for a complete automated analysis. The key work performed in this stage is to determine which bytes of the binary hold executable instruction and which hold data.

Since most recent architectures separate code and data, one may be tempted to believe the work at this stage to be trivial. That is indeed not the case because, while an architecture may enforce that a data segment does not contain code, there is no guarantee that the code segment does not contain data. Of course, since the code segment is write-protected, the data in the code segment can only be constant. Nonetheless, there can be data, or simply garbage, in the code segment.

The common method of disassembly, referred to as the linear sweep method, assumes that all bytes starting from the entry point of a binary (or some start location) are instructions, and disassembles the entire code segment, following successive bytes. When the content of some location does not match a valid instruction, one may assume that it is data.

This method is acceptable in a disassembler designed for interactive analysis of binaries, such as IDA Pro. However, it has obvious shortcomings when used for automated analysis.

A recently reported algorithm, called recursive traversal, overcomes some of the shortcomings (Schwarz and Debray, 9th Working Conference on Reverse Engineering, 2002). In this method code is disassembled by tracing the flow of control in the program. Thus, whenever a branch instruction is encountered the disassembly continues simultaneously at both the address following the branch instruction and the address that is the target of the branch instruction. Some targets that are reachable only through indirect control transfers are identified using a speculative disassembly process (Cifuente and Emmerik, IEEE Computer **33**(3), 2000).

Procedure abstraction

Once the executable instructions of a program have been identified, they may be segmented into groups representing procedures (or functions). This is motivated from the notion of procedures (and functions) in high-level programming languages. Since most compilers compile a procedure into a contiguous set of instructions, the procedure boundaries could be determined by identifying the entry points of successive procedures. The entry points in turn could be identified by identifying the CALL instructions (in the disassembly stage).

Unlike its high-language counterpart, a binary program does not have any construct identifying the beginning and end of a procedure. This problem does not appear to have been discussed in the literature and may need attention.

Control flow graph generation

A control flow graph (CFG) is a directed graph. Its nodes represent statements (or blocks of statements). Edges in the graph represent flow of control from one statement (or block) to another. A CFG is commonly created for a single procedure. Since a procedure in a high-level language has a single entry and a single exit point, it is common for a CFG to have a unique entry and exit node as well. In a CFG the node representing a procedure call may be linked to the CFG of the called procedure, thereby creating *interprocedural* CFG.

All static analysis techniques used in compilers and model checkers *assume* the existence of CFGs for the procedures of a program. Because a CFG is a language-neutral representation of the flow of control in a program, algorithms based on CFGs can be used for any (procedural) language.

An algorithm for constructing CFGs for high-level language programs is available in text books. The same method is adapted for assembly language programs.

Data flow analysis

There are various analyses that qualify as data flow analysis. The most common analysis is data dependence analysis, which is to determine the instructions that use the variable (register or memory location) modified by another instruction. The analyses performed for optimizing transformations are all classified as data flow analyses, or flow analyses.

Besides assuming the existence of CFGs, these analyses also assume that the data area of a program is segmented into variables, where each variable has a unique name (modulo scoping rules). Also associated to a variable is its type and size. A binary program has no such segmentation. The data area is simply a continuous sequence of bytes. Though the address of a data area may be treated as its name, the type of data it holds and the size (number of such units) is not obvious from the binary. The problem of identification of variables in a binary does not appear to have been discussed in the literature, and deserves attention.

Property verification

The property verification's phase determines the existence (or non-existence) of a property in a program. This phase takes two inputs: (1) a formal representation of the suspicious property (or properties) and (2) control flow graph and data flow information of the program. It outputs a determination whether or not the program satisfies the given properties. This answer is then translated into whether the program contains malicious behaviour.

For example, consider the property *SendsLotsOfMail* to be true if the following holds:

> The program contains a loop containing *GetEmailAddress* and *SendMail*.

In order to determine whether this property holds, the analyser may create a compacted CFG that contains only calls to *GetEmailAddress* and *SendMail*. It would then determine whether the two functions are in a loop.

Over the last decade the use of model-checking to verify the presence or absence of properties has gained prominence. Loosely speaking, model-checking is a way to check for the existence of a finite state machine (specification) in another finite state machine (program).

The property to be checked is described as a finite state machine that transitions on *atomic* predicates, properties that can be identified by a cursory look at the program. The program being checked is also converted to a finite state machine, created by abstracting away all the details except the atomic predicates observed in the program. Model-checking is then used to check whether a program has a given property.

PROBLEMS IN STATIC ANALYSIS OF BINARIES

We now discuss how a virus writer may attack the various stages in the analysis of binaries described above.

Attack on disassembly

In the von Neumann architecture there is no *definite* way to differentiate code and data that is resident in memory.

The linear sweep method can be fooled by introducing garbage data immediately after an unconditional jump instruction. The recursive traversal method can be fooled by placing garbage data after a conditional jump instruction and programmatically ensuring that the jump condition always succeeds. This will lead the recursive traversal algorithm to follow both paths, the instruction immediately after the conditional jump instruction and the target of the jump instruction – potentially leading to an inconsistent state.

To throw the disassembly off, the garbage data may be crafted so that it matches a valid instruction, thus beating a heuristic that validates the instruction after the jump instruction. The target of the jump instruction itself may be hidden by computing its address in a register and using an indirect jump instruction to transfer control to the address in that register.

Attack on procedure abstraction

Since a procedure is a basic unit of most analysis algorithms, a virus writer can defeat static analysis by making it harder to identify a procedure unit in a binary program. The analysis can be attacked if one cannot determine the boundaries of a procedure. This stage can be attacked by obfuscating the call instruction – say, by using a combination of push and jump instructions.

There is also no sanctity in the tradition of placing the code of a procedure in contiguous memory locations. In fact, this is just a tradition followed by compilers. There is no guarantee that hand-written assembly programs follow this tradition. The virus Win32.Evol is a classic example. Use IDA Pro to identify its procedures and you'd find that it beats the heuristic used by IDA Pro.

There is no necessity for such mangled code to be hand written. It will not be too hard to modify a compiler such as gcc so that it mangles the code for a procedure into non-contiguous blocks. Or such mangling can be performed automatically after an executable is created.

Attack on control flow graph generation

The construction of CFG can be attacked by fooling the CFG generation algorithm in creating redundant edges in the CFG. The creation of extra edges may jeopardize the precision of the analysis in the later stages. The CFG generation process can also be attacked by obfuscating the assembly code such that one cannot determine the correct target of a jump instruction, such as using a jump through register.

One method for resolving the targets is to assume that the jump will transfer control to every instruction (or to every block). This assumption, though safe for various static analyses for compiler optimizations, could spell doom for an anti-virus technology by increasing the time and space costs of the analyses.

Attack on data flow analysis

An example of attack on the flow analysis stage was discussed in the section entitled 'promises of formal analysis'. Most flow analysis algorithms propagate sets of data from one node of a CFG to another. When data flows into a node from two different predecessors, the two sets of data are merged to create a single set. In the process of merging the data some information is lost, leading to an imprecise analysis.

The data flow analysis stage can be attacked by moving some computation from a block (of a CFG) into the preceding nodes and obfuscating the computation along each path. The analysis can also be attacked by using data that resides outside the scope of the program, such as in the areas managed by operating system. The known constant values in these areas may be used in the program for computation, thus thwarting accurate analysis. For instance, Win32.Evol and other viruses utilize knowledge of the specific address where kernel32.dll is loaded. The same addresses could also be used to access constant data from kernel32.dll code.

Attack on property verification

Property verification is carried out using theorem-proving systems. Typically, these systems depend on human guidance to prevent them from getting into infinite loops. Completely automated theorem provers, such as model checkers, operate on an abstraction of the problem. Sometimes the abstraction itself may be so large that the theorem prover may take an inordinate amount of time and resources to complete the proof. Or else, the abstraction may throw away so much information that the theorem prover may yield results that are correct for the abstraction, but incorrect for the program being analysed.

A virus writer can attack the mechanism using the knowledge of how the theorem prover used in an anti-virus technology determines the existence of the *atomic properties* and how it composes atomic properties into more complex properties.

For instance, consider an anti-virus tool that uses the presence of calls to system library as atomic properties. This AV tool may look at the address of the target of a call instruction to determine whether a system library function is being called. Win32.Evol will escape such a virus detector because it obfuscates calls to system library functions. Instead of using the call instruction, this virus uses the return instruction to make the call. Before the return instruction is executed, though, the address of the function to be called is pushed on the stack. If the analyser cannot determine whether a virus calls a specific library function, the analyser has two choices. One, to assume that the program actually does not call the Win32 API, thus letting the virus pass through. Two, to assume that the program does call the library, thus generating false positives for programs that actually do not call the API.

CONCLUSIONS

We have argued that formal analysis methods used by optimizing compilers and other programming tools, though they appear promising in the detection of metamorphic viruses, are not directly suitable for use in anti-virus technologies.

The common approach for analysing a binary consists of the following stages: assembly, procedure abstraction, control flow graph generation, data flow analysis, and property verification. Compiler optimization-based methods for each of these stages can easily be attacked. Thus, even if the processing in each stage is correct 90 per cent of the time, the overall system will be correct only 59 per cent of the time, which is pretty close to the results offered by flipping a coin.

In order to use these formal analysis methods in anti-virus technologies, we may have to rethink the whole process. Unlike in the context of optimizing compilers and other similar tools, the analysis tool and the programmer (virus writer) do not have a common objective. Hence, assumptions made by analysis methods used by such compilers can be exploited by the virus writer.

The good news is that a virus writer is confronted with the same theoretical limit as anti-virus technologies. To keep ahead of anti-virus technologies, a metamorphic virus writer has to address some of the same challenges that anti-virus technologies face. This is likely to have an effect on the pace at which new metamorphic transformations can be introduced. It may be worth contemplating how this could be used to the advantage of anti-virus technologies.

PRODUCT REVIEW

SOPHOS MAILMONITOR FOR EXCHANGE 2000

Matt Ham

The *Sophos* product line is expanded regularly, and thus opportunities arise to review components of the wide-ranging suite of products available. On this occasion the subject of scrutiny is the company's *Exchange Server* product – once known as *SAV for Exchange* but now going by the name of *MailMonitor*.

The core on-access and on-demand scanners produced by *Sophos* differ little outwardly from those available two or three years ago. For this reason these components are largely skipped in this review. Similarly, reference is made to *SAVAdmin*, the administrative tool for rolling out *Sophos* products across a network – a review of this tool can be found in the October 2001 issue of *Virus Bulletin* (p.18).

Enterprise Manager offers functionality for, among other features, centrally managing updates and the source of these updates. The *Enterprise Manager* is still in a state of evolution with new features in beta which include detailed log file analysis. Also in beta is an *Exchange 2003 Server* version of *MailMonitor*. However, none of this beta material was examined in depth.

DOCUMENTATION AND WEB PRESENCE

The boxed version of the software supplied contained three slim booklets – the installation guides for *Windows 2000 Server*, *MailMonitor for Exchange* and *Windows NT Server*. In addition to these hard-copy installation guides for each component, there is also PDF documentation. This includes manuals for each product and such useful information as guides for updating the various components. Given that updating is such an important part of the life-cycle of anti-virus software, it was a disappointment that no hard copy version was supplied for this.

The collection of documentation is quite impressive in quantity – a book, 23 installation guides, 16 manuals, three supplements and advanced user-guides and three update guides being available in English. Versions of the documentation in German, Spanish, French and Japanese were also supplied.

The book included with the documentation is *Sophos*'s own *Computer viruses demystified*. This, along with a mouse mat, is included in the boxed product as a diversion from the purely product-related material. As a general overview of viruses the book is good for the genre, although outdated in parts. In particular, worms are afforded much less attention than they are currently due, with macro-viruses being considered the main threat to have gained ground with the ready availability of the web.

From an external viewpoint, the look and feel of the *Sophos* website (www.sophos.com) matches all the documentation and packaging. The home page provides access to the links for software updates, links to information on each major product line, news about recent viruses and more press release-style information. The ease of navigation is impressive given the volume of information on offer, and the technical FAQ pages are a particularly good resource. In addition, there is a facility to add *Sophos*'s virus information and hoax information statistics to third-party websites.

The English website is supplemented with mirror sites in German, Spanish, French, Italian and Japanese. This reflects the international expansion which *Sophos* has wholeheartedly embraced over recent years.

INSTALLATION

The *Sophos* product CD autoruns, giving the option to install products as one of its features. Installation from CD has proceeded smoothly in the past when installing *Sophos* products for comparative reviews, and it was assumed that a simple click would suffice here, as it had done on so many occasions before. However, the *MailMonitor* installation process is definitely one for which reference to the manuals is required.

I launched the installation application, only to be confronted with an error due to *SAVI* not being present. It is not mentioned that the presence of *SAVI*, the *Sophos Anti-Virus Interface*, is vital until numerous dialog boxes have passed – and indeed there is no mention of what *SAVI* is or how it can be installed. The need to have an existing installation of *Sophos Anti-Virus* is mentioned, although no link is stated between this requirement and the *SAVI* error. *SAVI* is, in fact, the central core of the *Sophos* scanning engine which provides APIs to other applications – in this case the *Exchange* product does not contain the scanning engine, so must be able to interact with an existing *SAVI* installation.

The manuals were consulted and it became apparent that the *Server* product must be installed first – the recommended method being by the use of a central installation. This installation was carried out, the lack of *SAVI* persisted and the manuals were consulted once more. Performing central installation does not, in fact, install the software but merely transfers files so that installation can be performed from this repository of files. The installation has to be performed manually after the file transfer to the central installation directory (CID) – which is not explained at all on-screen during installation.

The process of installing from the CID is not an automated one, but rather involves a degree of poking about in the nether regions of the server. First, an update user must be created. This is explained in the manual, although dialogs that are not mentioned in the manual appeared on the machines with, for example, *Exchange Server* installed upon them. Default settings were used in these cases.

Following this user addition, local security policy must also be edited so as to allow the update account to log on as a service. Although, admittedly, the administrator should end up being more aware of changes on the server as a result of having to set these parameters manually, it does seem unwieldy to be forced to use manual methods where automation could have been an option.

With the CID present, and the update account set up and ready for action, the installation proper can commence. Again, this is a more manually-intensive method than might be expected, with the approved method being to use the start-run dialog and browse to a long path name. The dialog which results is initially identical to that produced by the central installation procedure, though the choices involved diverge rapidly.

File locations were left as the defaults – and other choices were made with constant reference to the manual (since, by now, this seemed a better choice than relying on assumptions). The *InterCheck* client, the on-access portion of the software, should not be installed on the *MailMonitor* machine. Thus *InterCheck* support was not installed on this occasion. This configuration option was mentioned as being important in the autorun dialogs, though not in the *Windows 2000* server installation guide – however this documentation has since been upgraded.

Less contentious options chosen were to activate automatic updating from the CID and to allow removal by users – since in this case the user was likely to have good reason to do so. Both of these were default settings.

Activating auto-update required the decision as to whether updates should be with or without user interaction. Without user interaction there is also the facility for a user to delay auto-updating. Auto-update by default checks for new functionality every ten minutes. This can be changed to any arbitrary number of minutes, within the equally arbitrary limits of 5 and 1439, daily or weekly. At this point a précis of the installation options is displayed, prior to full installation.

When installation was complete the machine started instantly on an unannounced scheduled scan – which had not been mentioned anywhere in the installation procedure. Scheduled scanning of this nature was disabled during the testing, so as to provide a platform where only user-initiated scanning could occur.

After all this preliminary activity, the ground was prepared for the installation of *MailMonitor*. The Administration Console and mysteriously named 'services' may be installed at this point and both were selected by default. In contrast with the manual addition of users for the central installation, the *MailMonitor* service account is created as part of the installation process. Similarly, a group of administrators is created for *MailMonitor*. The process of file transfer is completed after this – no reboot being needed at any stage.

However, there were still some steps involved in installation. The exclusion of the temporary and quarantine folders from scanning is the final step and this must be performed manually. Since the temporary directory and the quarantine directory are placed by the installation routine, it is astonishing that this procedure is not automated. (As a related issue, the quarantine directory was not created at this stage when upgrading from those past *MailMonitor* versions which used a quarantine mailbox.)

UPDATING

The CID installation is the source of updates for those products linked to it. These may be clients or, in this case, the server-based *SAVI* installation. With this in mind it becomes clear that updating is a matter of updating the CID rather than individual machines – a much less onerous position in most real-world situations.

There are two methods of update available: manual and by use of *Enterprise Manager*. *Sophos Anti-Virus* is updated on a monthly cycle, with a new version available on that timescale. Between versions there is a constant stream of virus definition updates. This schedule of updates leads to the particular way in which data is stored and updated.

Where manual updates are selected there are three areas which must be considered when updating a CID. The CID includes two repositories of program files, one for *Windows NT, 2000* and *XP*, while the other contains *Windows 95, 98* and *ME* data. Although the *SAVI* installation on *Windows 2000* requires only the former information for use by *Exchange*, partial updating can hardly be recommended. Each month these repositories become totally superseded by new versions and must be replaced. The third information type stored are the IDE files by which *Sophos* transmits new virus identities to its software. Clearly, each month will see an accumulation of such files which will be redundant at the next full application update. These must therefore be deleted when a new program version is available, to be replaced gradually as new IDEs are made available over the following month.

The methods for performing these updates are fairly simple, in that the CID can be reinstalled when a new version is

produced, this process updating the main program data without the need for any of the associated configuration editing required when installing the CID initially. In the case of IDEs, the process is even simpler, consisting of deleting the appropriate files.

With this simplicity in mind the usefulness of *Enterprise Manager* (*EM*) to automate matters might be questioned. However, automation is obviously a great boon when multiple CIDs are considered and even more so when IDEs are considered. Since IDEs are released at unpredictable intervals, with no respect for day or night, automating their installation is far more efficient and pleasant for an administrator than performing the same task manually.

Enterprise Manager operates as an MMC snap-in, which sports a logo rather reminiscent of that used by *GDATA* in its products. The initial interface is simple indeed, consisting of two large buttons for creating and opening a library. 'Libraries' are the *EM* term for areas where packages of data which will form updates are stored. Initially the only option that will work is that to create a new library. Selecting this option gives the choice as to where files will be stored, after which the library is created. Once created the library must be populated by the user.

The packages used to populate the library are supplied by *Sophos*, though the route may be direct or indirect. In order for packages to be downloaded the parent source must be defined, which is either a website or a UNC path. Since it is possible that one source will be unavailable, a secondary parent may be defined for use if the primary parent cannot be contacted. One criticism that might be levelled here concerns the extreme use of mixed metaphors, with parents having packages as children who are stored in a library. Be that as it may, it offers automated updating and this allows many sins to be forgiven.

In addition to its update features, *Enterprise Manager* also offers installation over a network. This is, however, performed through initiating *SAVAdmin* rather than as a dedicated function. It should also be noted that CDs are not supplied with any of the *EM* packages, and thus use of a UNC location for updates does not allow *EM* to update via the CD distribution of *SAV*.

Since the majority of testing was performed on machines isolated from the outside world, thorough testing of download speeds for the package distribution site was not investigated.

FEATURES OF MAILMONITOR FOR EXCHANGE 2000

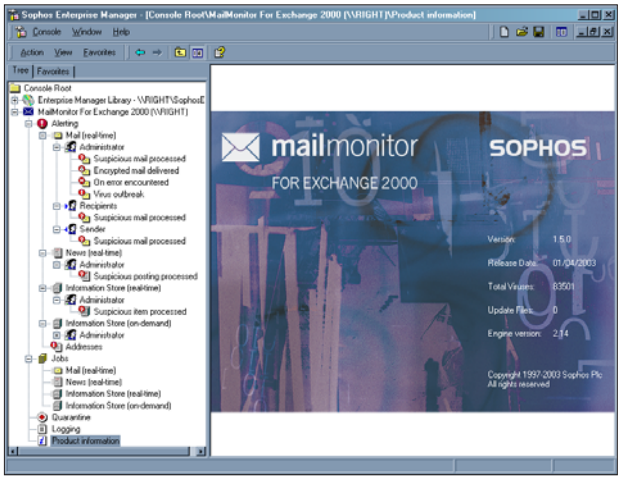
Like *EM*, *MailMonitor* is an MMC snap-in. The snap-in must be loaded within MMC before it can be used. This is simple enough, as is the manual creation of a shortcut to the *MailMonitor* functionality. As before, however, automation would seem preferable – these actions are, after all, automated for *EM* when that snap-in is installed. It was also noted that after upgrading the version of *MailMonitor* on the test machines, a reboot was required.

The interface for *MailMonitor* is as might be expected of a MMC snap-in – that is a right-hand view pane, the contents of which are determined via the tree in the left-hand pane. The main tree branches allow access to the Alerting, Jobs, Quarantine, Logging and Product Information panes.

The alerting pane is that with the largest number of sub-branches, due to there being configurable alerts for Mail, News and both on-demand and real-time scans of the Information Store. For all but mail these alerts are reserved for the administrator, for informing of the presence of suspicious objects. Where mail is concerned the sender and recipient may also be selected as targets for being informed that a suspicious object has been detected.

The administrator is also offered a larger selection of alert types, with encrypted mails, those mails with errors in them and the presence of 'virus outbreaks'. The default trigger for a virus outbreak alert is for five viruses to be detected in one minute. Once this threshold is reached the outbreak alert is sent and, by default, alerts are terminated until a further threshold is reached. By default this threshold is set as no viruses in a five-minute period. This is clearly an area where large and small organisations are likely to vary in their settings and these settings may even be required to be adjusted on a rolling basis.

The use of the term 'suspicious' is somewhat at odds with the standard descriptions in the alerts – all objects which can cause an alert are deemed worthy of being called viruses. This may be an area where customisation is the



order of the day, and that is available for subject, body and, where appropriate, each object which is suspicious.

Moving on from the alerting options, the jobs branch allows access to the scans which produce these alerts. The real-time scans of mail, news and information store all share a common format. Each has tabbed views, with Status, Scan, Actions and Logging being present in all three cases. Status is a general overview of that particular scan, with some statistics on suspicious objects, errors and the like. Scan is where details of the scanning method are set.

By default only the most likely infected areas are scanned in each file, archives are scanned and Macintosh viruses are searched for. The omission of scanning files fully has no effect upon any virus known in the wild, though the faster scan rates it produces do result in misses on a small subset of zoo viruses. Logging can be directed to Event Log, File Log and screen, the default here being that all events are recorded in each of these areas.

Actions to be taken upon detection of suspicious objects are set individually depending upon the classification under which the infection falls. Infections in signed and unsigned mails may thus be treated differently and failed disinfections treated as special cases; a wide selection of other cases are also catered for.

Where mail files are concerned, two further tabs are available, one for use in determining email addresses where mail will not be scanned for infection, recipients and senders being individually configured. It is also possible to add tags to the subject line of scanned mails and this is configured in its own tab. Information Store scanning also has an extra tab, this being the choice of which store to scan, if more than one is present.

The on-demand Information Store scans follow the same tabbed format as those already discussed, though multiple scans may be configured. These can also be scheduled or limited to specific mailboxes, allowing for more directed searches than those allowed by the real-time scans.

Quarantining is the default fate of any infected object which passes through the *Exchange* server. The information available on each infected object includes time of detection, sender, recipient, subject, infections, the scan type which detected the object, size of the object and file name.

Suspicious objects may be sorted by each of these parameters and blocks selected for treatment in a variety of ways. The files involved may be disinfected, attachments deleted or the entire message deleted. Deletion of attachments is not quite the whole story, however, since in testing this deleted only attachments which were infected. If either disinfection of attachment removal has been selected the quarantined message may thereafter be delivered to its intended recipient. Samples may also be sent directly to *Sophos* from the Quarantine interface.

As mentioned previously, logging may be directed to screen, event log or file log. Of these the screen log is available within the individual scan branches. The on-screen version is likely to be of secondary importance in comparison with the logging to file, however.

File logs are collated by type in the logging branch of the interface. The interface provided here allows browsing to and viewing of individual logs, and files may also be deleted or limited in size. No further filters may be applied, which is slightly disappointing. Likewise statistics are limited to simple numeric values of files infected, numbers quarantined and the like. The most recent beta version of *Enterprise Manager* does, however, go some way to remedying this.

The Tree view on the left has a parallel view allocated to favourite functionality. This is a standard MMC feature, but useful in that *EM* and *MailMonitor* are both MMC snap-ins. This allows creation of combined functionality between the two, despite being distinct and separate applications.

CONCLUSION

It will have become apparent in the course of the review that the installation of *MailMonitor* and its associated companion applications is perhaps the most irksome part of the whole procedure. Once installed, updating is catered for in an automated fashion and the day-to-day working activities seem relatively pleasant to perform. To a certain extent the fragmentation of the installation process is paralleled by the large number of product-specific manuals. This concentrates information and makes it an easier task to determine which manual should be consulted at each stage, rather than having to fumble through a monolithic tome. Though this is probably a good, or at least neutral, effect, it tends to be overshadowed by the large number of stages at which the administrator is forced to perform configuration manually.

Technical details

Test environment: All machines used in this test were identical 1.6 GHz Intel Pentium 4 machines with 512 MB RAM, 20 GB dual hard disks, DVD/CD-ROM and 3.5-inch floppy drive.

Server software: *Windows 2000 Server Service Pack 2* with *Exchange 2000 Server Service Pack 2* and *Outlook 98*.

Client software: *Windows XP Professional* with *Outlook Express*, *Windows XP Professional* with *Outlook 98*.

Developer: Sophos Plc, The Pentagon, Abingdon Science Park, Abingdon, OX14 3YP, UK. Tel +44 1235 559933; fax +44 1235 559935; web www.sophos.com.

END NOTES & NEWS

The 10th International Computer Security Symposium, COSAC, takes place 14–18 September 2003 at the Killashee House Hotel near Dublin. A choice of more than 40 sessions and six full-day master classes is available. For full details of the agenda, venue, travel discounts, partner programme and registration see http://www.cosac.net/.

COMDEX Canada 2003 will be held 16–18 September 2003 in Toronto, Canada. Discounted registration fees apply until 22 August. For details of the conference, exhibition and keynotes see http://www.comdex.com/.

The 13th *Virus Bulletin* International Conference and Exhibition (VB2003) takes place 25–26 September 2003 at the Fairmont Royal York hotel in Toronto, Canada. Full details, including conference programme, abstracts and information about the venue can be found on the *VB* website. Register online at http://www.virusbtn.com/conference/.

The 5th NTBugtraq Retreat takes place in the days immediately following the *Virus Bulletin* conference in Ontario, Canada. A welcome event on the evening of 26 September will be followed by the Retreat from 27–29 September 2003. Full details can be found at http://www.ntbugtraq.com/party.asp.

Black Hat Federal 2003 takes place 29 September to 2 October 2003 in Washington D.C. For more information and online registration see http://www.blackhat.com/.

InfowarCon 2003 takes place 30 September to 1 October 2003 in Washington D.C. Military leaders, representatives of political forces, academics and industry members will discuss the concepts of the latest ongoing initiatives in the Homeland Security and Critical Infrastructure Protection communities. For details see http://www.infowarcon.com/.

The Fifth International Conference on Information and Communications Security (ICICS2003), is to be held 10–13 October 2003 in Huhehaote City, Inner Mongolia, China. For full details see http://www.cstnet.net.cn/icics2003/.

The Workshop on Rapid Malcode (WORM) will be held 27 October 2003 in Washington D.C. The workshop aims to bring together ideas, understanding and experience relating to the worm problem from academia, industry and government. See http://pisa.ucsd.edu/worm03/.

COMPSEC 2003 will be held 30–31 October at the Queen Elizabeth II Conference Centre in Westminster, London, UK. This year's conference will include the Compsec 2003 Poster Session, featuring a review of the latest scientific advances in computer security research and development. For full details see http://www.compsec2003.com/.

The European RSA Conference will be held 3–6 November at the Amsterdam RAI International Exhibition and Congress Center, The Netherlands. For details of the agenda, location and registration see http://www.rsaconference2003.com/.

The Adaptive and Resilient Computing Security (ARCS) workshop will take place 5–6 November 2003 at the Santa Fe Institute, NM, USA. The aim of the workshop is to stimulate novel approaches to securing the information infrastructure. In particular the workshop will consider long-term developments and research issues relating to the defence of information networks. For full details see http://discuss.santafe.edu/bnadaptive/.

AVAR 2003 will be held on 6 and 7 November 2003 in Sydney, Australia. The theme for the conference is 'Malicious Code', incorporating emerging malicious code threats, the technologies at risk and the technology needed to deal with these threats both now and in the future. See http://www.aavar.org/.

COMDEX Fall 2003 takes place 15–20 November 2003 in Las Vegas, USA. See http://www.comdex.com/.

The Infosecurity.nl exhibition takes place 11–12 November 2003 at Jaarbeurs complex, Utrecht, Netherlands. For all the details, including information on how to participate, a list of exhibitors and floorplan, see http://www.infosecurity.nl/.

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