MARCH 2006

virus BULLETIN

The International Publication on Computer Virus Prevention, Recognition and Removal

CONTENTS

IN THIS ISSUE

PROXIES FOR SALE

The Locksky email worm is one of the increasing number of profit-oriented malicious programs to which we are becoming accustomed. Ferenc Laszlo Nagy takes a look at one of its many variants.
page 4

VECTOR SWITCH

Two years after its emergence the Beagle family is still one of the most pervasive families of Internet worms. John Canavan takes a close look at one variant that has made the surprising switch from email to ICQ as its major infection vector.
page 7

DYNAMIC TRANSLATION

Dynamic translation is a technique that can be used instead of emulation for decrypting complex malware. Adrian Stepan describes how the technique can also be used to perform generic unpacking.
page 11

vbSpam supplement

This month: anti-spam news & events; fighting spam with data compression models

ISSN 1749-7027

'Our jobs ... are still as much about people and what they do (and don't do) as they ever were.'

Max Morris
Independent author, USA

VIEW FROM THE CHEAP SEATS

I am often asked what I like about my job as director of threat intelligence services for a large banking organisation in the US. I always answer the same way: it is never dull and if you wait a week, a day, or even an hour, what you think you know and the threats for which you are prepared will always change. We are in an industry where we face a constantly evolving challenge, and one that is not for those who want an 8-to-5 job, or who need a lot of sleep.

When I think back to how the threats, industry and the tools we use have evolved, it is interesting to realize how much, in fact, has stayed the same. Although the attack vectors have become increasingly sophisticated (remember the good old days when we could simply tell users not to open emails from people they didn't know?) and the number of vulnerabilities continues to grow while the time between discovery and exploitation decreases, our jobs and our day-to-day successes and failures are still as much about people and what they do (and don't do) as they ever were.

The good news is that our tools have matured, are more widely used, and as a result, the level of protection has improved. Consumers have made strides in recognizing the importance of security and companies have dedicated more resources to developing strategies and implementing solutions to address the problem. The bad news is that, even with all of the progress we have made, it still only takes one infected laptop connecting to a company's internal network to elude all perimeter defences, or one keystroke logger to steal someone's personal confidential information and access their financial accounts.

We still fight the battle of social engineering and wrestle with people's natural curiosity and trusting natures. The same tactics that made the I Love You mass mailer and the Nigerian 419 scams successful are employed today – and still reaping the same benefits. We still deal with software that contains vulnerabilities, and have to face the complexity of patching and the struggle to get people to understand why they need to patch. When you ask users whether they patch their systems regularly, it is not uncommon to be met with a look of confusion.

We still work with the reality that, more often than not, while vendors have made strides in releasing pattern files faster, developing improved heuristics, incorporating behaviour detection and being more responsive in providing patches for new vulnerabilities, we respond reactively, not proactively, to new risks and threats.

Some things have changed. While there is still a prevalence of mass mailers that continue to be a common vector for propagation, we are seeing movement towards a stealth-like approach and more targeted attacks. In many cases adware and spyware have replaced malware as the more pressing issues to be addressed. And the type of attacker has evolved from script kiddies to perpetrators motivated by financial gain.

What does this mean for the people who use the Internet, the companies whose businesses rely on the software packages that continue to contain vulnerabilities, and you and me, the people in the trenches whose day-to-day job is to protect systems and data? As complicated (and varied) as the answer can be, at the most basic level it is also very simple. We must continue to do what we have been doing: finding ways to improve our tools, improve our communications and think outside the box.

And while defence-in-depth layered protection, product enhancement and innovation will go a long way towards continuing to provide us with improved security at home and in business, we must keep in mind that there is no magic bullet staring at us from over the horizon and, most importantly, it will take all of us continuing to work together to fight the good fight.

As we take a collective deep breath, after having been awake for 20 straight hours, and realise that tomorrow is yet another day full of challenges to be faced, just ask yourself: would we really have it any other way?

NEWS

HOTBAR ADWARE DISPUTE SETTLED

Symantec has reached an out-of-court agreement in the pre-emptive lawsuit it filed against marketing firm *Hotbar.com Inc.* In the unusual case, the anti-malware vendor sought a court ruling that would allow it to label certain *Hotbar.com* products as adware (see *VB*, July 2005, p.3). Under the terms of the settlement *Symantec* will dismiss the suit, but it will continue to classify *Hotbar*'s program files as low-risk adware.

UPDATING NIGGLES

Last month proved to be troublesome for security vendors *Sophos, Microsoft* and *Kaspersky*, as niggles with updates caused problems for their customers.

Sophos customers suffered an onslaught of false positives thanks to a fault in the update file which was released to add detection of the OSX-Inqtana-B worm for Mac OS X. The fault resulted in *Sophos Anti-Virus* generating false alerts on a number of files in *Microsoft Office 2004* and *Adobe Acrobat Reader*. A revised update was released shortly after developers spotted the problem, alongside an apology to customers.

Meanwhile, many of the users of *Microsoft*'s *Antigen* email security product were left without fully functional email systems for several hours after they received a faulty update to the *Kaspersky* scanning engine. The *Antigen* product – which *Microsoft* inherited when it acquired email security firm *Sybari* last year – uses a number of different scanning engines including *Kaspersky*'s to provide anti-virus protection. A *Microsoft* spokesperson explained: 'As soon as we were aware that our customers were experiencing email problems due to the *Kaspersky* update, we escalated through the appropriate channels across *Kaspersky* and *Microsoft* and were able to define, test and provide a resolution.'

Indeed, *Microsoft* did not have an easy month at all with its security products – just days before the problems with *Antigen*, an update to *Windows AntiSpyware beta 1* caused it to misidentify *Symantec* security tools as password-stealing malicious software. On detection of certain registry keys set by the *Symantec* products, *Windows AntiSpyware* generated an alert and prompted the user to delete the keys. Users who went ahead and deleted the keys would have found that *Symantec AntiVirus* and *Symantec Client Security* software stopped functioning correctly.

Fortunately for the two companies, only a small number of customers are thought to have been affected by this error, due to the fact that the misidentification applied only to *Symantec*'s enterprise products.

Prevalence Table – January 2006

Virus	Type	Incidents	Reports
Win32/Sober	File	1,656,867	88.38%
Win32/Netsky	File	79,090	4.22%
Win32/Mytob	File	75,607	4.03%
Win32/MyWife	File	33,227	1.77%
Win32/Mydoom	File	10,748	0.57%
Win32/Bagle	File	4,683	0.25%
Win32/Zafi	File	3,960	0.21%
Win32/Lovgate	File	2,716	0.14%
Win32/Sdbot	File	2,499	0.13%
Win32/Funlove	File	1,233	0.07%
Win32/Brepibot	File	500	0.03%
Win32/Valla	File	431	0.02%
Win32/Klez	File	263	0.01%
Win32/Bugbear	File	233	0.01%
Win32/Feebs	File	226	0.01%
Win95/Spaces	File	203	0.01%
Win32/Bagz	File	198	0.01%
Win32/Pate	File	197	0.01%
Win32/Dumaru	File	188	0.01%
Win32/Gibe	File	186	0.01%
Win32/Mabutu	File	179	0.01%
Win32/Reatle	File	129	0.01%
Win32/Mimail	File	111	0.01%
Win32/Maslan	File	90	0.00%
Win32/Loosky	File	82	0.00%
Win32/Dref	File	73	0.00%
Win95/Tenrobot	File	73	0.00%
Wonka	Script	67	0.00%
Win32/Bobax	File	65	0.00%
Win32/Chir	File	59	0.00%
Redlof	Script	46	0.00%
Win32/Elkern	File	46	0.00%
Others[1]		535	0.03%
Total		1,874,810	100%

[1]The Prevalence Table includes a total of 535 reports across 69 further viruses. Readers are reminded that a complete listing is posted at http://www.virusbtn.com/Prevalence/.

VIRUS ANALYSIS 1

PROXIES FOR THE UNDERWORLD: I-WORM.LOCKSKY.AS

Ferenc Laszlo Nagy
VirusBuster, Hungary

The Locksky email worm first appeared in October 2005 as a successor to Trojan.Tofger. The aim of both Tofger and Locksky is to install proxies on victims' machines, which can then be sold. The prices of the proxies are listed on a website (http://proxy4u.ws:8080/) which also provides two ICQ contact addresses for prospective buyers – one for home users and small businesses, and the other for clients looking to purchase more than 500 proxies.

VERSIONS AND VARIANTS

The first version of Locksky was disguised as an Internet phone utility named 'Skylook' (from which the worm's name was derived). Several new variants have appeared since then, mainly incorporating changes designed to help the virus bypass anti-virus detection.

The most active variant at the time of writing is I-Worm.Locksky.AS (internal version number 0039). The following analysis will cover this variant, but most of what is described here also applies to the previous versions. Also note that while I refer to internal version numbers, these may not be unique – for example, there are two variants with version number 0034: I-Worm.Locksky.AC and I-Worm.Locksky.AI.

THE ENCRYPTOR

The main file and six dropped components are all encrypted by a unique encryption routine. The decryptor is added to the end of the last section. The new entry point is the same as in the original program. The code starts with an anti-emulation routine using MMX instructions, then jumps to the decryptor at the end of the file.

The decryptor code itself is encrypted by a long chain of elementary operations. These operations can be considered to be part of the key, and change in every instance.

The (decrypted) decryptor then writes the original (but encrypted) 80 bytes to the entry, decodes the whole encoded part and relocates the section if needed. Through the decoding, every 32-bit value is xored by a constant key, then incremented by 1.

Interestingly, we have seen a small number of seemingly unrelated pieces of malware (such as TrojanSpy.Banker.CAU and Trojan.DL.Tibs.I) encrypted in a similar way. It is likely that these are malware files that have been developed independently, then encrypted and distributed by the group responsible for writing the Locksky worm.

DROPPER COMPONENT: SACHOSTX.EXE

This is the main file that arrives in the Locksky-infected email. Its icon is a white arrow on a red box, and in this version it is named ebay_info.exe.

If the operating system is not *NT*-based, the file exits immediately. Otherwise, it checks the system folder for 'hard.lck' and deletes it if it finds it, thus causing any older versions of the worm to exit. Then it enables network access for hard.lck – which seems to be a bug. To continue cleaning up after older versions, it then terminates processes where the name of the exefile contains 'sachostc', 'sachosts', 'sachostw' or 'sachostm'.

Next, it copies itself to the Windows folder as sachostx.exe, and sets the 'HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Run\HostSrv' registry entry to this file, thus ensuring that the file will run every time the machine is booted.

Next, it drops a keylogger component named msvcrl.dll and activates it by calling the hide_get and hook_kbd functions. It also drops and runs a password collector named sachostp.exe. This will write the initial content of the file SYSDIR\attrib.ini.

Finally, it creates the hard.lck file and enters the main loop. One responsibility of this loop is to watch hard.lck, and terminate the worm if it disappears. The other is to wait for an Internet connection and perform various tasks (described below) once there is one.

First, it drops and runs the sachostw.exe worm component. If the computer is connected directly to the Internet, it also drops the sachostc.exe and sachosts.exe proxy servers with a random port as the parameter. (Port numbers must be larger than 10,000, and different for the two programs.)

All three programs require network connections, so the worm opens the firewall by running the 'netsh firewall set allowedprogram <filename> enable' command for them. The programs are not executed directly, but the worm loads the Windows system file svchost.exe, injects the starter code into its address space, and runs it. On *Windows 2000* (where svchost.exe is a console application), this will be manifested in console window popups.

UPDATING AND REPORTING

If the current time is 10 minutes past the hour, the worm will try to update itself. It tries two web servers,

proxy4u.ws:8080 and usproxy2u.ws:8080, the first of which currently works. It checks the advertised version number of the update, and if it is greater than its own version number (in this case 0039), it will download and run the new executable. It is no surprise that the downloading itself is performed by svchost.exe. If the file runs successfully, the worm exits.

If the spy components (keylogger msvcrl.dll and password collector sachostp.exe) have collected enough information (attrib.ini is larger than 150,000 bytes), the worm drops sachostm.exe, opens the firewall for it, and runs it through svchost.exe.

The worm also reports back to its authors. If the machine is behind a proxy (as reported by the InternetGetConnectedState API function), then every five hours, if it is connected directly to the Internet, it downloads the following URL every 10 minutes: 'http://proxy4u.ws/index.php?IP=%u.%u.%u.%u&Port1=%d&Port2=%d&ID=%x&Ver=%s&con=%s&speed=%d', where:

- IP: victim IP address
- Port1: sachosts.exe port number
- Port2: sachostc.exe port number
- ID: unique ID created from machine name and disk volume serial number
- ver: worm version number (0039 in this case)
- con: connection type (P: proxy, L: LAN, M: modem, U: unknown)
- speed: download speed in kilobytes/sec (downloads the www.microsoft.com home page, and divides the number of downloaded bytes by the number of whole seconds elapsed, if that is not 0).

PASSWORD COLLECTOR COMPONENT: SACHOSTP.EXE

This component writes the passwords it finds to 'attrib.ini'. If started from *Windows 9x* it will write passwords returned by mpr.dll's undocumented WNetEnumCachedPasswords function. If started from an *NT*-based system (which is always the case when started by the Locksky worm), it collects data from the registry and from the Protected Store (P-Store).

In the registry, subkeys below the 'HKCU\Software\Microsoft\Internet Account Manager\Accounts' key are investigated. If the 'HTTPMail User Name' entry (which contains *Hotmail* account data) exists, then it is logged together with the 'HTTPMail Password2' value. If the 'POP3 User Name' entry exists, then it is logged along with 'POP3 Password2'.

The Protected Store COM object is also queried for the following categories (as named by the worm):

Deleted OE Account	IE:Password-Protected sites
OutlookExpress	MSN Explorer Signup
IE Auto Complete Fields	AutoComplete Passwords

SPY COMPONENT: MSVCRL.DLL

This key and web traffic logger also operates as a rootkit. It implements four functions: hook_kbd, un_hook_kbd, hide_get and un_hide_get.

The hook_kbd function installs a GetMsgProc hook, which logs the keypresses and text placed on the clipboard together with the title of the active window and current time. These are all written to attrib.ini.

The hide_get function tries to hide the processes whose names start with 'sachost', and the registry entries named 'hostsrv'. This is achieved by hooking CBTProc and patching NtQuerySystemInformation and RegEnumValueW functions in every process to which it is mapped while servicing CBTProc messages.

By patching ws2_32.dll in memory, it filters network traffic and logs GET and POST HTTP requests and the answers returned by the server if the URL contains one of the following keywords:

abbey	bank	barclay
cahoot	egg	e-gold
forex	halifax	hsbc
ktb	lloyds	log
mail	money	nationet
nationwide	natwest	nwolb
openplan	passport	password
PayPal	rbs	secret
secure	sell	sign
woolwich		

Additionally, all POST requests are logged.

The un_hook_kbd and un_hide_get functions uninstall the mentioned routines. The operations performed by msvcrl.dll can make the system very unstable, and the rootkit function does not work in most cases.

INFO MAILER COMPONENT: SACHOSTM.EXE

If the machine is not currently connected to the Internet, it waits for a connection by sleeping in a loop. Then it sends

the content of attrib.ini in an email to the address kuraser@list.ru using the smtp.list.ru SMTP server. (This is one of the few changes from the 0034 version, where the email address was erwaderrruio@list.ru.)

The file will be base64-encoded, and named '00000000-00000000.txt', where the first number is made from the machine name and disk volume serial number (by a different algorithm from that in the main file), and the second number depends on the number of clockticks since power on.

Finally, it deletes attrib.ini and exits.

WORM COMPONENT: SACHOSTW.EXE

When started, the worm component will copy sachostx.exe (the main file, which contains all parts) to its own directory (the SYSDIR) as temp.bak. Then it waits for an Internet connection if necessary.

To simplify base64 encoding, 1 to 3 bytes are added to the worm copy, so the number of bytes will be divisible by 3. This means that the worm will become longer in every generation. The termination of base64 encoding is not standards compliant (CRLF is missing before the boundary line), so some MIME extractors may have difficulties with it.

Further characteristics of the Locksky-infected emails are as follows:

- The subject line is 'Your Ebay account is suspended' appended with spaces – possibly because one can update it this way without recompiling.
- The message body (also space padded) is:

```
Dear eBay Member,

We regret to inform you that your eBay account
could be suspended if you don't re-update your
account information.

To resolve this problem please see details in
attached file.

If your problems could not be resolved your
account will be suspended for a period of 24
hours, after this period your account will be
terminated.
```

- The dropper component is attached as ebay_info.exe.
- The boundary is always 'zl'.

Messages are sent to three groups of addresses:

- The worm will search for *.htm* files in the administrator's documents (the folder to which the 'HKLM\Software\Microsoft\Windows\CurrentVersion\Explorer\Shell Folders\Personal' registry entry points), and collect the addresses by searching for "mailto:xxx" strings, where xxx is the email address.
- It gets the value of 'HKCU\Software\Microsoft\WAB\WAB4\Wab File Name' and reads email addresses from the WAB file by interpreting the file format.
- It reads 'The Bat!' mailer configuration from the registry (keys below HKCU\Software\RIT\The Bat!), and searches for lines containing 'To: ' in the file messages.tbb.

It will use the SMTP and sender settings defined in the 'HKCU\Software\Microsoft\Internet Account Manager\Default Mail Account' registry folder, in order to lend the 'From' addresses some authenticity. The SMTP connection is managed by the worm's engine.

Finally, it deletes temp.bak and exits.

PROXY COMPONENTS: SACHOSTC.EXE AND SACHOSTS.EXE

These components are fully-featured proxy server applications. They may have been developed by others, but they have been stripped down to become perfect for malware purposes.

Sachostc.exe is a SOCKS5 proxy. It supports username/password authentication, but no password is compiled in it. It supports connect, bind and udp association modes.

Sachosts.exe is an HTTP proxy. It supports CONNECT and generic requests.

With the help of these components, the infected machines can be instructed remotely to connect to given target hosts and carry out actions such as downloading web pages or sending email messages.

CONCLUSION

This worm is one of the profit-oriented malicious programs to which we are becoming accustomed. Instead of advertising, its authors chose a more direct way of making money: selling the availability of infected machines.

There is nothing revolutionary in this worm's code, and its development had been limited to trying to prevent it from being detected by anti-virus products.

Regarding the future, we have just captured a variant with version number 0044 – although this one does not have a worm component. It seems as if this spreading method may have generated too much attention from the anti-virus field, and that the authors have switched back to producing Trojans as a result. Alternatively, the switch could be a transient idea and there might be more aggressive variants to follow – who knows?

VIRUS ANALYSIS 2

IM_A NUISANCE – W32.IMAV.A

John Canavan
Symantec Security Response, Ireland

Two years after its emergence, the Beagle family is still one of the most pervasive Internet worms.

Leading the way in the new modular malware model, the Beagle family of threats has thrived by breaking its functionality down into separate basic components. Its associated downloaders, email address harvesters and many other Trojan parts have flourished, growing their network ever further and making their venture a profitable one.

The first variants of the Beagle family were seen in January 2004, and from the outset it used some interesting techniques that had not typically been seen in mass-mailers.

The initial mass-mailing samples opened a backdoor on infected machines, listening on TCP port 6777. This allowed a file to be downloaded and executed on the system when a trigger string was sent to the backdoor. It also ran a notification thread which contacted a remote website announcing the presence of a newly infected system. This notification would prove to be a key technique which the authors used to keep track of their infected network and to seed new variants and components.

Later variants of the worm terminated security-related processes by deleting their associated registry keys and files, and stopped and removed system services. They overwrote host files to prevent access to security-related websites and even uninstalled previous Beagle variants. They did most of this while injected in explorer.exe.

For a period, the Beagle authors engaged in viral warfare against the authors of the Netsky family. Beagle variants would terminate Netsky-related processes automatically and delete their registry keys, then create their mutexes to prevent re-infection. Most of these features and the design methods employed focused on keeping control of the infected machine – allowing easy installation of updates, hiding the presence of the infection of the system and making disinfection more difficult.

Given this background, it is hard to imagine the motivation behind what appeared to be a major change in focus in one of the latest variants of the family to surface: the switch to ICQ as its major infection vector.

W32.IMAV.A

First sighted on 26 January 2006 at known Beagle Trojan download URLs, W32.Imav.A appeared at first to be just another Beagle Trojan.

Downloaded with the filename my_foto.zip, Imav's first course of action was to display a candid photograph of a flame-haired, freckle-skinned girl in a green bikini hanging out by a lighthouse.

It then set about copying itself to %System% and dropping its associated dll file alongside it.

MD5	Size	FileName
960dddec022cc846a0a0075b98906c7b	33,745	im_1.exe
e2562b406a7cdf53ed50adfcf2f9fcd9	17,886	im_2.exe

When executing from %System%, Imav adds the value "im_autorn" = "%System%\im_1.exe" to the registry Run key. It then starts up the usual Beagle Trojan routines, proceeding to kill a list of security-related processes, polling another list of URLs for a file to download, disabling a list of security-related services, renaming or deleting a long list of filenames from security-related products and some associated registry entries. All pretty standard fare.

On a little further inspection, however, it became evident that Imav had more to offer.

ICQ

Since Eric Chien and Neal Hindocha presented their paper 'Malicious threats and vulnerabilities in instant messaging', at VB2003, anti-virus analysts have awaited the emergence of a rapidly spreading IM worm. Until now we have seen attempts to propagate via instant messaging from Kelvir, Bropia, Funner, Bisex, various IRC bots and many more, but none have succeeded to a level comparable with the classic mass-mailing email worm. So, perhaps when one of the most successful of the classic email worms turns to IM, we should be worried.

Throughout its history Beagle has adopted new techniques designed to increase its effectiveness and ability to spread. It makes sense that its authors would attempt to make use of whatever infection vectors are available, and therefore the inclusion of an IM module is not a surprise.

What is strange is the particular means of IM propagation Beagle's authors chose. By restricting themselves to *ICQ* they immediately lost a huge section of potential users, but they cut their potential infection pool yet further by making use of a password-stealing technique which is unique to *ICQ Lite/2003* versions of the software. Not only that, but if the Public Mode of these versions of *ICQ* is chosen on install, the password will not be stored in the registry and thus the worm rendered impotent.

These choices appear even more curious when we look at PWSteal.LdPinch. A variant of this Beagle-related Trojan was downloaded by versions of the Mitglieder Trojan in early 2004. It logged keystrokes and sent system

information on to a remote address, but interestingly also had the ability to steal *ICQ* passwords. LdPinch had the ability to steal passwords from client versions *ICQ99b-2003a/Lite/ICQ2003Pro*, reading and decrypting each *ICQ* profile's MainLocation value from the registry, but could also retrieve passwords stored in the .dat file used by older versions of *ICQ* and it even stole passwords from alternative *ICQ* clients *Trillian*, *Miranda* and *&RQ*.

W32.Imav.A iterates through *ICQ* profiles stored in the registry at:

HKEY_CURRENT_USER\Software\Mirabilis\ICQ\NewOwners\<UIN>\

HKEY_LOCAL_MACHINE\Software\Mirabilis\ICQ\NewOwners\<UIN>\

A number of versions of *ICQ* store the user's password here in the value MainLocation. This value is encoded based on the volume serial number of the system; however decryption techniques are well-known and have been used by several other malware authors (Bizex, Ldpinch) to date.

HUNGRY? HAVE A SNAC

What is particularly noteworthy is that Imav.A communicates directly with login.icq.com, logging itself in as a client. Imav builds its own FLAP packets of SNAC data. (SNAC is the basic communication unit that is exchanged between clients and servers – the SNAC communication layer sites on top of the FLAP layer.)

To log into the server Imav needs to re-encrypt the password it has just decrypted from the registry. This is done with a simple byte-for-byte xor with the following array:

```
0xF3, 0x26, 0x81, 0xC4, 0x39, 0x86, 0xDB, 0x92, 0x71,
0xA3, 0xB9, 0xE6, 0x53, 0x7A, 0x95, 0x7C
```

The FLAC login packet is constructed as below, with the Type-Length-Values specified.

```
4 BYTE     0x00 0x00  0x00 0x01
 TLV(1)    STRING     UIN/ICQ Number
 TLV(2)    STRING     Encrypted password
 TLV(3)    STRING     Client Version, "ICQBasic"
 TLV(16)   WORD       unk, 0x010A
 TLV(17)   WORD       major version, 0x0014
 TLV(18)   WORD       minor version, 0x0020
 TLV(19)   WORD       lesser version, 0x0000
 TLV(1A)   WORD       build version, 0x090B
 TLV(14)   DWORD      version, 0x0000043D
 TLV(0F)   STRING     language, 2 chars, "en"
 TLV(0E)   STRING     country, 2 chars, "us"
```

Once authenticated, Imav connects to the BOS server, using the host/port information and auth-cookie from the login.icq.com's initial reply, and completes the protocol negotiation stage.

When the login process is complete, Imav attempts to send messages to random users containing the string 'my foto' and a link to my_foto.zip located on a remote server.

IP FILTERING

Another interesting routine used by Imav is its blocking of security-related websites. Older variants of Beagle made use of a technique that is commonly seen in the malware world. They added a list of the hosts they wanted excluded to the windows hosts file %System%\drivers\etc\hosts. Although effective against the average home user, this was easily spotted and could be rectified simply by checking the contents of that file.

In *Windows 2000*, *Microsoft* introduced an API to implement packet filtering functionality. The API allows for similar functionality to that included in the TCP/IP properties of a network adapter. To impede the user further in removing it from their machine, Beagle now makes use of this Packet Filtering API to drop packets destined for a pre-defined list of websites. GetAdaptersInfo() returns a linked list of completed IP_ADAPTER_INFO structs, from which the worm can get the current IP address assigned to all active interfaces (PIP_ADDR_STRING CurrentIpAddress;).

PfCreateInterface() creates a new filter interface. This new interface will be used to control the adding and deleting of filters from the adapters retrieved from GetAdaptersInfo(). The filter is created with the default PF_ACTION_FORWARD attributes for its inAction and outAction. The new filter interface is then associated with each of the active network adapters using PfBindInterfaceToIpaddress(). At this point the packet filter is active and in place, but not set to filter anything. Imav performs a DNS lookup on each site to be blocked and creates an associated PF_FILTER_DESCRIPTOR struct for each result. This struct defines the packet filter containing details of the source and destination to filter on.

PfAddFiltersToInterface() then adds the filter to the previously created filter interface. The filter reverses the default processing rule for the interface, that is, the rule that was specified during the call to PfCreateInterface(). So, in this case traffic to hosts matched by a filter rule will be dropped. Imav sets both input and output filters for the PF_FILTER_DESCRIPTORS generated.

CONCLUSION

Although the impact of this variant in the wild was minimal, its use of new techniques reminds us that the Beagle authors will continue to be a threat as they pursue new means of propagation. As they embrace this experimentation, we must keep a close watch on developments.

FEATURE 1

HOW CAN A WEB FILTER ADD PROACTIVE SECURITY?

Steen Pedersen
Equant, Denmark

It is often recommended that administrators deploy several layers of security across their systems. This is in order to maximise the chances of being able to prevent a threat from entering and executing in the environment.

In this article I will explain how the outermost layer of defence, the perimeter defence, can be enhanced.

THE LAYERED APPROACH

The layers of protection can be separated into five levels within the IT environment, starting from the outside and moving inwards:

1: Perimeter

2: Network

3: Host

4: Application

5: Data

In simple terms, client anti-virus solutions work with signatures and generic detection techniques to stop new and unknown threats.

New threats are becoming increasingly difficult to detect – and these days the threats we see are not only viruses and worms, but also Trojans, backdoors, spyware and other malware that is seeded through email, instant messenger and websites. Every sample can be unique, making detection harder for conventional anti-virus solutions.

Personal firewalls combined with host intrusion prevention systems provide additional proactive protection. However, the client protection is the third layer of defence and it is preferable to block threats as early as possible – ideally before they reach the workstation.

Email gateway solutions have become quite effective in blocking new and unknown threats. This is achieved by spam filtering, the use of blacklists and by blocking potentially harmful email attachments. The same protection can be applied to web communication by using a gateway scanner to scan HTTP and FTP traffic for malware.

PERIMETER DEFENCE

The perimeter is the first point of defence. Solutions deployed at this level include firewalls, VPN connectors and gateway scanners for FTP, HTTP and SMTP.

URL filters and web filtering applications are often overlooked as perimeter defence solutions because they tend to be thought of as solutions for enhancing productivity, rather than security.

However, the use of these applications can be a powerful defence against malware since they enable administrators to understand, monitor and control outbound web access. This level of control allows administrators to keep their users away from the 'bad neighbourhoods' on the Internet and prevent them from visiting sites that are known to contain threats.

NON BUSINESS-RELATED WEBSITES

The majority of threats on the Internet are located at websites that would not generally be classed as business-related. Sites that contain pornography, illegal music, movies, games and software, gambling, P2P/file sharing, hacking and other inappropriate content are notorious for playing host to spyware and other malware.

A web filter can be used to prevent access to such sites, which in a business environment will increase both productivity and available bandwidth. In addition to this, there are security benefits to blocking access to these sites, as both known threats and – more importantly – new, unknown threats (that would not be detected by anti-virus software) are prevented from entering the computing environment.

As well as blocking both known and new malware, the web filter can also be used to identify systems that are already infected. This can be achieved by monitoring web activity – infected systems will display unusual levels of activity.

TECHNOLOGY AND POLICY

When implementing web filter solutions one must consider the trade off between improving security and restricting access to the Internet. It is important that, while preventing the damage that can result from visiting unauthorized websites, the web access policy does not cause too many problems for the users.

The web filter should be configured in monitoring mode to begin with. After a period of time and regular reviews of the filter reports the configuration can be changed to 'advisory mode', where the user must confirm access to non business-related sites.

The technical solution must be supported by a prudent web access policy that is defined and supported by upper management. As well as a policy for web access, procedures for handling blacklisting and whitelisting need to be implemented and communicated to the end users.

A web filter can enforce a very strict policy which allows access only to whitelisted sites. However, this might not be very practical in the real world.

Another solution would be to create a list of non-categorized sites that are visited. This list can be reviewed regularly and approved sites added to the whitelist. All non-categorized sites remain blocked until the sites are whitelisted in the web filter or categorized by the vendor of the web filter database.

HOW SECURE?

An important point to bear in mind is that a web filter is not an 'install and forget' solution. Even with the best filtering database a web filter alone cannot provide full protection. The administration and handling of the web filter are very important parts of the solution and resources must be allocated for this. New sites pop up, sites can change content and sites can be categorized wrongly.

A web filter will certainly reduce security problems and provide information about where some of the breaches originate. In combination with the web filter, the firewall configuration and monitoring of the firewall log is also a key to solid perimeter defence.

LOOK AT THE PAST AND LEARN

It can be useful to review what has happened in the past and use the experience to make changes that will improve security. By monitoring and logging web activity we can collect information that will be useful for enhancing perimeter security.

For example, we can use the logs to determine which Internet sites are visited most frequently by users, and the category to which those sites belong.

We can also find out more specific information, such as which internal user/system generates the most web communication to non-categorized sites, which non business-related sites are the most visited and which non-categorized sites are the most visited.

This information may highlight new and unknown sites which could be the cause of security problems either now or in the future. It can also pinpoint particular users whose Internet activity might raise security concerns. Detective work like this should be included in an ongoing procedure to improve the web filter and the level of security it can provide.

A report combining local anti-virus alerts and web filter log information can also reveal important details. For example, if virus alerts are generated on a particular user's machine and the location of the infected file is always in the browser cache directory, this is an indication that the user is visiting insecure websites. Information about the websites visited at the time of the virus alert can be found in the web filter log file.

BLOCKING FILES BASED ON FILE TYPE

Web filters can also control when (and if) a user can access or download specific file types on the Internet (such as .pif, .com, .lnk, .vbs or .exe). This feature is very similar to the email attachment blocking rules which are often implemented on SMTP gateways and mail servers.

The risk of downloading and activating malware can be reduced significantly by blocking access to these specific file types. However, the number of companies using web filters to block the downloading of unwanted file types is still very small compared with the number of companies using email attachment file-blocking rules.

CONCLUSION

With threats evolving, increasing in volume and becoming more sophisticated, traditional firewall and anti-virus solutions alone are no longer sufficient to protect our systems. We must consider what other solutions can be used to enhance the level of security.

Web filters are often overlooked as security solutions, but by implementing a layered approach and adding proactive solutions at the different levels (perimeter, network, host, application and data) we can create a formidable defence. In addition to increasing security, this can also enhance productivity and bandwidth, and reduce the amount of time spent on handling security incidents.

We can make a change. Instead of spending time on security incidents, administrators should invest time in handling and maintaining the proactive security layers.

FURTHER READING

[1] Mitchell Ashley, StillSecure, 'Layered Network Security 2006: A best-practices approach'.

[2] *Secure Computing*, Technical Paper, 'Spyware: an annoying and dangerous problem you can eradicate' and 'Protecting HTTP traffic: Why web filtering should be your first line of defense'.

FEATURE 2

IMPROVING PROACTIVE DETECTION OF PACKED MALWARE

Adrian Stepan
Microsoft, USA

Malware writers have always striven to create malware in such a way that it will evade detection by anti-virus software. Code obfuscation is one of the methods they use to achieve this. Over time, obfuscation techniques have evolved from simple encryption to polymorphism, metamorphism and packing. The latter is becoming increasingly popular these days, with lots of packing tools available to malware writers, and more of them being created each month.

A lot of developmental effort is required for anti-virus engines to provide unpacking support for all these packers in a timely manner. Generic unpacking is one solution to this problem, but implementing it is not a trivial task. The amount of computing power required by some unpacking routines is beyond the amount that most emulators can provide in a reasonable time. Is there any way to overcome this limitation?

At the VB2005 conference last year I presented a paper entitled 'Defeating polymorphism: beyond emulation'. The paper described a technique called 'dynamic translation' (DT), which can be used instead of emulation for decrypting complex polymorphic malware. The main advantage of this method over emulation is that it is significantly faster. Dynamic translation can also be used to perform generic unpacking, with very few or even no changes required.

OF OLD AND NEW OBFUSCATION TECHNIQUES

Theoretically, there is no limit to the complexity of the algorithm that a malicious program can use to obfuscate its code. In practice, however, the complexity of such algorithms is limited both by the fact that the malware in question needs to replicate in a reasonable time, and by the amount of effort that malware writers are willing to invest.

Recently, the majority of new malware has been written in high level languages (HLL), most likely because development is fast and today's malware writers lack assembly skills. Writing polymorphic viruses in a high-level language is very difficult though, and as a result their prevalence has dropped significantly. However, compression algorithms can be developed easily in HLL, and these can be used to pack almost any executable file.

In the case of polymorphic viruses, the number of different encryption algorithms used is so large that writing a specific decryption routine for each of them is not feasible. In the past, however, generic decryption by means of emulation was easier to implement thanks to several factors:

- Decryption algorithms required only a few instructions for decrypting each byte, for all but a handful of viruses.
- Viruses were quite small, ranging from less than a hundred bytes to a few kilobytes in size; decrypting a virus required emulation of relatively few instructions (from a few hundred to a couple of million).
- The amount of memory required was typically small, about the same order of magnitude as the size of the virus.
- Decryption typically used only simple arithmetic/logic instructions, so the emulators didn't have to support the full x86 instruction set. FPU/MMX/SSE instructions were rarely used, and (with very few exceptions) they did not affect the decryption process and could be skipped.

On the other hand, the number of different packers/protectors available is small – a few hundred, only a couple of which are polymorphic (e.g. Morphine, Molebox). The number of different compression algorithms used by these is even smaller (Huffman, arithmetic/rangecoder compression, LZW, BurrowsWheeler and a few variants or combinations of these). Therefore, specific unpacking routines to handle all of them can be developed with a reasonable amount of effort.

Algorithms used in compression are more complex and have more stringent memory requirements than the encryption algorithms used by polymorphic viruses. Modern malware, especially those written in high level languages, are also a lot larger than we have seen in the past (up to several hundred kilobytes). Therefore, unpacking them requires many more instructions and proportionally more computational effort.

The use of emulation to achieve unpacking is possible, but it may take a considerable amount of time to complete and it is difficult to implement. Static (specific) unpacking is currently the most widely used unpacking method in AV engines, for both flexibility and performance reasons.

FROM SPECIFIC TO GENERIC UNPACKING

The use of packers is becoming increasingly popular among malware writers. A packed piece of malware has a better

chance of remaining undetected for a long time, as well as spreading faster due to its smaller size. Packing an existing piece of malware is also by far the easiest way to create a 'new' one. Of the new incoming samples we see, more than 50% are produced simply by repacking existing malware, using different packers.

Presently, there are a few dozen different packing utilities available, most of which can be used by anyone with minimal computer skills. Each of these tools may have several variants or versions – for instance, UPX has more than 20 known versions. There are several scrambler tools that can be used to create modified versions of UPX.

Source code is available on the Internet for a number of packing tools, such as UPX, FSG, Yodacrypt and Morphine. These can easily be modified by any malware writer, by tweaking the compression algorithms or by adding one or even several encryption layers. Such modified packers are being created at a rate of about 10–15 per month.

There is no reliable method of determining whether a given file is packed. Running all available unpacking routines on all files would make scanning unreasonably slow. Therefore, a preliminary check needs to be performed for each supported packer, to determine if a given file might be packed with it. These precheck routines need to be fast, and usually rely on searching for a code pattern known to be present in the unpacker. However, a simple change in the unpacker code would prevent identification of the packer, meaning that unpacking would not even be attempted. In other instances, a packer detection routine may identify a modified packer, but the unpacking routine will not work correctly, due to the modified compression algorithm, added encryption or other changes.

It is possible to write unpacking routines that are resilient to significant variations, but not without more development effort and most likely with some speed penalty as well. In reality, unpacking routines are modified 'after the fact' to accommodate new packer versions. The use of generic unpacking would significantly improve proactive detection capabilities while requiring less development effort in the long term, as more and more packers need to be supported.

OVERCOMING THE SPEED LIMITATION

Emulation has successfully been used in AV engines for a long time to achieve decryption of polymorphic viruses in a generic way. Although emulation can be used for unpacking as well, the method may be too slow to be effective in practice. Typically, emulation is several hundreds of times slower than execution, achieving a speed of about 10 MIPS on an average PC. Large packed files or files packed multiple times may need to run several hundreds of millions or even billions of instructions to be unpacked. Using an emulator for unpacking such files can take several minutes, which is not acceptable for an AV engine.

Emulation speed is not the only difficulty that needs to be addressed in order to perform generic unpacking. Emulators require large amounts of virtual memory, good support for the CPU instruction set, accurate emulation of exception handling, timing and synchronization mechanisms, support for multi-threading, etc. While these problems can be addressed by investing development effort in improving an emulator, the slowdown is a limitation inherent to the method, and one that is very difficult to overcome.

Dynamic translation serves a purpose that is very similar to emulation; however the implementation follows a different principle. An emulator 'interprets' a program's code instruction by instruction. Each instruction is first decoded, in order to determine the instruction type, length, operands, etc. After this, the emulator identifies and calls a routine that updates a data structure describing a virtual system, in the same way as the original instruction would change the state of the real system if executed. These emulation routines are generated statically at 'compile-time' and are part of the emulator.

With dynamic translation, the first step is to divide the program to be analysed into 'basic blocks', that are contiguous blocks of code having a single entry point at the beginning and a single exit point at the end of the code. After each block is identified, it is translated to native executable code, in a process similar to just-in-time compilation. The resultant code is persisted and then executed. A block will only need to be retranslated if the original code is overwritten; this only happens if the program uses self-modifying code – a technique rarely used in programs written in high-level languages. Code that is executed several times, such as a loop construction, will be translated and persisted at the first loop iteration, and for all subsequent iterations the persisted code will be executed.

The method provides a significant speed advantage over emulation, by eliminating the redundant analysis of repeating code sequences. This is especially true in the case of an unpacking routine, which is in fact a loop that uncompresses a few bytes of code at each iteration. An additional increase in speed can be achieved by omitting detection of self-modifiable code for programs written in HLL.

SOME SPECIFIC ASPECTS OF GENERIC UNPACKING

Running a packed binary inside a virtual machine, based on either emulation or DT, will produce an unpacked image of

the binary, assuming the virtual machine is implemented properly. However, this does not guarantee detection of a piece of malware that is present in the obtained image, as in most cases, the image will not be identical to the original binary file. This happens because several packers use specific techniques to modify the original binary, in addition to compression, for the purpose of defeating generic unpacking.

A possible mitigation is the use of signature patterns or detection routines that are resilient to these techniques. Another solution relies on reconstructing the original binary from the image obtained by using generic unpacking. This can be achieved by identifying the obfuscation techniques that have been used and running appropriate specific or generic routines to revert the effect of those techniques.

A few packers, such as Pespin, redirect API calls by changing the offsets of indirect call instructions in the packed program to use an address table generated by the unpacking routine. Pespin also uses a 'code stealing' technique to obfuscate some code sequences in the original program. A particular code chunk (starting, for instance, at the original entry point) is selected and removed. In order to maintain the original functionality, an equivalent code sequence is generated at a different address and called instead of the original code. Any signature including the stolen code sequence or modified call offsets will no longer be matched for a packed malware. Detection can be achieved by using dedicated routines to recover the original code, or by not using any stolen/modified bytes as part of a signature.

In some cases, files that are packed multiple times with different packers will no longer be functional. For instance, files that are packed with Molebox and after that packed again with another packer will fail to execute correctly, because Molebox relies on the code integrity of the file to compute a key needed for unpacking. In this case, generic unpacking, based on either emulation or DT, will fail. Specific unpacking routines may still succeed to unpack such a file, by computing the key based on a partially unpacked image of the file immediately before the Molebox layer.

CONCLUSION

Dynamic translation can be used to achieve generic unpacking with good speed performance. However, detecting generically unpacked malware using an existing signature set is not guaranteed to succeed. New signatures can be extracted to mitigate this problem. Combining DT with specific routines to rebuild the original binary from an unpacked image is yet another solution to be explored.

COMPARATIVE UPDATE

CHANGES TO THE VB TEST SETS

Matt Ham

This month's reviewer activity has been concentrated behind the scenes, with some labour-intesive changes being made in the test sets used for *VB*'s comparative reviews.

CLEAN TEST SET

Over the years *VB*'s clean test sets have consisted of a reasonably representative selection of files. Recently, however, there has been some concern over the inclusion of a large number of dynamically compressed files. In most cases these are installers, which contain multiple executables under potentially proprietary encryption or compression algorithms. While a small number of these would be expected in everyday on-demand scanning, the test sets contain a far larger percentage than one would expect to encounter in a real-world situation.

With the number of scanners that contain routines for delving into such files on the increase, it has become apparent that such in-depth investigation has a severe impact on speed of scanning. Therefore, the inclusion of a large number of installer files in the clean test sets was putting the more thorough products at a disadvantage in terms of the scanning speeds we reported.

To resolve the situation, the clean executables test set has been split into two. One set contains 'pure' executables, while the other contains these dynamically compressed executables. It is hoped that this will enable a better breakdown of any future scanning speed issues.

SPYWARE TEST SET

The second major change comes in the form of a new test set – a spyware test set. Currently, there are no plans to make the detection of samples in this test set a requirement for achieving a VB 100% award.

While recognising that it is not the most stringent of test methodologies, the current plans are not to look at spyware detection capabilities on machines that are already compromised. Instead, it is envisaged that the files included in the test set will be the initial vector of the malware in question. Thus the downloaded file of a spyware application, Trojanised software with spyware functionality, backdoor servers and the like will make up the bulk of samples in the test set. As the compilation of both the spyware test set and testing methodology is still a work in progress, I would be very pleased to receive comments and suggestions at matthew.ham@virusbtn.com.

TECHNICAL FEATURE

SOLVING THE METAMORPHIC PUZZLE

Rodelio G. Fiñones
Fortinet Technologies, Canada

Richard T. Fernandez
Trend Micro, Philippines

Metamorphic viruses have posed a challenge for the anti-virus industry for quite some time. There are several different techniques that can be used by metamorphic viruses to obfuscate their code – from simple register swapping to the more complex heavy code mutation. This article focuses on a number of metamorphic techniques used by 32-bit viruses under the *Windows* environment and highlights different methods for detecting them.

1001 WAYS TO DO IT

Over the years, viruses have demonstrated a number of obfuscation techniques to escape detection by anti-virus scanners.

Encryption is the simplest form of code obfuscation. In this technique, a constant key is used to decrypt the encrypted virus body. The virus uses the same key to encrypt all the files that it infects. The decryptor is placed at the start of the virus code and the encrypted data follows. However, since the decryption code is constant, it can be used to recognise the virus, making detection relatively easy.

Oligomorphic viruses differ from simple encrypted viruses because they have a varying, but finite number of decryptors in every infection generation. The method used for detecting simple encrypted viruses cannot be applied to this type of virus because the decryptor is not constant. Instead, the virus can be decrypted on-the-fly. This is achieved by capturing modifications made in consecutive memory locations – a characteristic of viruses that decrypt their virus body.

Polymorphic viruses produce a nearly infinite number of new decryptors with a variety of encryption methods to encrypt the constant virus code for every infection. Since the virus code remains constant after decryption, the detection solution used for oligomorphic viruses can be applied.

However, a more sophisticated method for detecting these viruses is through the use of a 32-bit emulator, which allows virus codes to execute in a controlled environment. The virus codes are then monitored and examined periodically, especially when certain instructions modify portions of the code. However, emulation uses a lot of memory operations and CPU resources. Another technique, called x-raying, allows us to see through the layers of encryption. Using this method, one can acquire the decryption key and decrypt the virus part that needs to be matched. This is applicable if the encryption algorithms are finite or have certain weaknesses.

Of all the code obfuscation methods that exist today, metamorphism is the most difficult to deal with. A good description of metamorphic viruses is given in [1]. Metamorphic viruses have neither a decryptor, nor a constant virus body. However, they are able to create new generations that look different. They do not use a constant data area filled with string constants but have a single code body that carries data as code.

UNMASKING THE CULPRIT

The evolution of metamorphic viruses in the Win32 environment makes the life of the anti-virus researcher a little more challenging. Some metamorphic viruses avoid storing strings in their normal form to prevent easy detection. In this scenario, scan string and range scanning are no use as detection techniques. Instead, techniques such as file structure analysis, code analysis, and behaviour analysis must be used.

As mentioned in [1], in order to detect a metamorphic virus perfectly, a detection routine must be capable of regenerating the essential instruction set of the virus body from the actual instance of the infection. [1] also introduces some of the techniques used to detect metamorphic viruses. In this article, we will enumerate and discuss in detail how each detection technique works and elaborate on their corresponding advantages and disadvantages.

FILE STRUCTURE ANALYSIS

File structure analysis, also known as geometric scanning, involves detecting the modifications that are made by the virus in the structure of the victim file. Some anti-virus experts also call this method ‘shape heuristics’, owing to the fact that it does not ensure an exact detection and is prone to false alarms.

Win32 binary viruses commonly rely on infection markers to flag files that have already been infected, thus avoiding multiple infections. For example, in the case of Bistro.B, an infected file has a high-byte value 0x51 at the minor linker version field. Such infection markers can also be very useful from the point of view of detection, since they can be used as initial filter signatures, narrowing down the number of files to be scanned.

However, a filter signature alone is not sufficient to ensure error-free virus detection. A better approach is to combine geometric scanning with other detection techniques.

WILDCARD * STRING AND HALF-BYTE SCANNING

Wildcards and the half-byte detection technique can be used to detect simple metamorphic techniques such as register swapping and op-code changing. Let's look at an example. Figure 1 shows a code fragment of a Regswap infection.

```
BE04000000      mov     esi,000000004 ;" ?"
8BDD            mov     ebx,ebp
B90C000000      mov     ecx,00000000C ;" ?"
81C088000000    add     eax,000000088 ;" ê"
8B38            mov     edi,[eax]
89BC8B18110000  mov     [ebx][ecx]*4[00001118],edi
2BC6            sub     eax,esi
49              dec     ecx

BB04000000      mov     ebx,000000004 ;" ?"
8BCD            mov     ecx,ebp
BF0C000000      mov     edi,00000000C ;" ?"
81C088000000    add     eax,000000088 ;" ê"
8B30            mov     esi,[eax]
89B4B920110000  mov     [ecx][edi]*4[00001120],esi
2BC3            sub     eax,ebx
4F              dec     edi
```

Figure 1: Regswap infection code fragment.

The parts shown in bold are the common virus codes for every generation. These are good candidates for a detection pattern. Half-byte detection would be appropriate for this type of infection if the scanning engine supports it.

STACK DECRYPTION DETECTION

A new metamorphic technique emerged when variants of the Zmorph virus appeared. In this case, a piece of polymorphic code is positioned at the entry point of the infected file. This decrypts the virus one instruction at a time and rebuilds it by pushing the result into the stack memory. After the last instruction has been decrypted, control is transferred to the start of the virus body, which is also located in the stack memory.

In order to detect this type of metamorphism, emulators must be able to detect stack decryption. The emulator must monitor the memory that is accessed by the virus. Once control is transferred to the stack memory, the emulator detects it and dumps the whole decrypted virus code for identification.

Note that monitoring the memory locations accessed by the virus while emulating has a significant impact on performance and thus filter checking should be applied first.

SUBROUTINE DEPERMUTATION

Another level of metamorphism was introduced when Win32 viruses such as Ghost and Zperm were released. Here, the virus code may be constant but metamorphosis is achieved by dividing the code into frames and infections – these frames are positioned randomly and connected by branch instructions to maintain the process flow. The diagram shown in Figure 2 illustrates this.

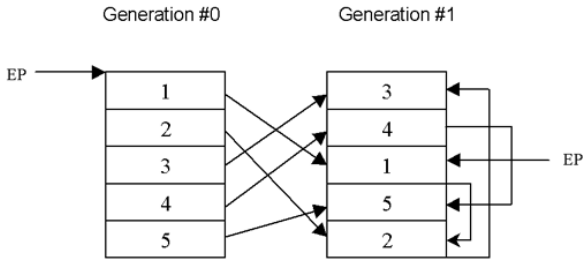

Figure 2: Subroutine depermutation.

The branch instructions could be a simple relative jump (0xe9, 0xea, 0xeb) or a complex transfer of control (i.e. push val32; ret). As shown in Figure 2, the control flow remains the same for the two infections.

The level of permutation varies depending on the number of subroutines that constitute the whole virus. For instance, the code of a virus that has eight subroutines can mutate by eight or 40,320 ways. The level of permutation can be computed as n (where n refers to the number of subroutines/frames). To make detection more difficult, most viruses insert garbage instructions between frames.

The Zperm virus employs a Real Permutating Engine (RPME) to accomplish its sophisticated levels of metamorphism. To counter this method, we need to perform partial emulation (emulation of branch instructions only) to reconstruct the virus code in its form prior to permutation. Figure 3 shows the process of rebuilding the permutated virus code.

The challenge here lies in deciding when to stop decoding and in ensuring that the virus codes are thoroughly exhausted. With the help of the decode table and IP address table, this can be done easily. This technique can be very effective for rebuilding the code as well as removing garbage.

DUMMY LOOPS DETECTION

An 'improved' version of Bistro was released some time after the original. In addition to an RPME engine, it has another anti-emulation technique: the random code insertion technique (aka macho engine). It inserts do-nothing instructions and dummy loops randomly before the decryptor codes. As a consequence, some emulators fail to rebuild the real virus codes, instead emulating millions of do-nothing instructions.

To avoid this problem, an emulator must have a means of identifying do-nothing instructions and dummy loops and must be able to skip them as encountered.

Permutated code	Decoding procedure
aaa1	1. decode aaa1
aaa2	2. decode aaa2
aaa3	3. decode aaa3
jmp @A	4. change IP to @A
bbb1	5. decode aaa7
bbb2	6. decode aaa8
@B: aaa4	7. decode aaa9
aaa5	8. change IP to @B
aaa6	9. decode aaa4
jmp @C	10. decode aaa5
bbb3	11. decode aaa6
bbb4	12. change IP to @C
@A: aaa7	13. decode aaa10
aaa8	14. decode aaa11
aaa9	15. decode aaa12
jmp @B	16. decode ret
@D: aaa13	
aaa14	
ret	
bbb5	
@C: aaa10	
aaa11	
aaa12	
ret	

Figure 3: Permutated virus rebuilding process.

In the case of Bistro, the macho engine can be detected by monitoring the movement of IP and checking the 'WRITE' operations. Generic detection of all Win32 viruses that utilize macho engines is possible using this method. However, a drawback of the method is that it is also susceptible to false positives.

REGULAR EXPRESSION AND DFA

One of the most efficient ways to deal with different types of obfuscation is the use of disassembly code to match the pattern (regular expression) using Deterministic Finite Automata, or DFA.

In its simplest terms, a regular expression is a formula for matching strings that follow a pattern. It provides a mechanism for selecting specific strings from a set of character strings. DFA is a transition table containing states and their corresponding next states.

Before digging into the details, we must define some terminologies:

- Automaton – a predetermined sequence of operations. In this context, it corresponds to the sequence of disassembly codes.
- Grammar – the rules for a language. In this context, the grammar pattern pertains to the collection or set of disassembly codes that the virus uses and provides the rule or the positive filter for detection.

As an overview, this detection method simply treats the virus file as a series of disassembly codes (alphabets) that can be matched against a database of existing virus disassembly codes.

In this technique, the scanning of a file is terminated automatically when the current disassembly code does not match any of the disassembly codes in the database or when the disassembly code does not belong to the acceptable list of instructions for a certain virus. Thus, this solution is relatively fast compared to others.

Two main components are involved in this solution: the builder and the simulator. The builder creates the automaton of the virus using the grammar pattern, while the simulator performs the automaton matching and conditional test using RegEx operators during file scanning.

The grammar pattern contains information on normalization (a set of garbage or negative filters) as well as information on how to detect the malicious file (Grammar and Accepted instructions). It uses regular expression where each item represents an assembly instruction.

An opcode can be any Intel IA-32 assembly instruction and an operand can be any of the following:

- Exact – specifies the exact operand to match.
  For example:

```
PUSH    EAX
```

- Wildcard – specifies the general type of the operand.
  For example:

```
PUSH reg32
MOV reg, imm
```

  Note that for the first assembly line, the PUSH instruction must be present with any 32-bit register. The next instruction requires that the MOV opcode is present with any register as the first operand and any immediate value as the second operand.

- Variables – information on an operand may be stored in a variable and retrieved later for matching.
  For example:

```
DEC     reg32_varset1
PUSH    reg_var1
```

  Note that while matching, the DEC opcode must be present in the first assembly line with any 32-bit register as the operand and set register variable 1 to this register type. For the next line, the PUSH opcode must match and the operand register must also match the retrieved value of register variable 1.

In wildcard instructions, the opcode and the operand vary. Possible values for the register operand are REG, REG8,

REG16 and REG32, while the possible values for the immediate operand are IMM, IMM16 and IMM32. For memory operands, MEM, MEM16 and MEM32 are the possible values. Assembly instructions are associated through operators such as start (*), plus (+), qmark (?), or (|) and explicit dot (.).

As shown in Figure 4, the pattern source format is processed by the DFA builder to produce automatons. Each assembly instruction is assigned a unique ID for easy matching and added to the corresponding garbage, accept and grammar list. Since our pattern is composed of operators, it has to deal with precedence. For easy processing, the pattern can be converted from infix expression to postfix expression before creating the DFA patterns.

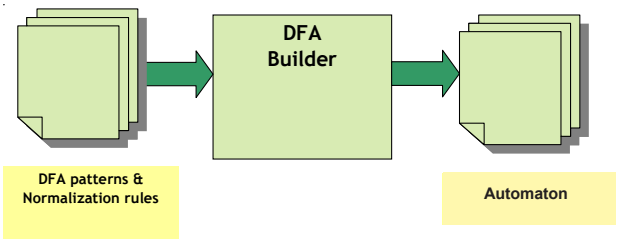

Figure 4: The DFA building process.

The simulator is responsible for scanning files for malicious content. It has four sub-components: a disassembler, depermutator, normalizer and DFA simulator. The input data is pre-processed by the first three sub-components before they are passed to the DFA simulator. Figure 5 shows the simulator component.

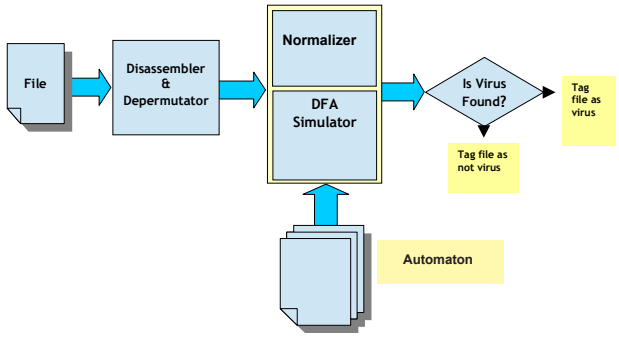

Figure 5: The DFA simulation process.

The disassembler part performs the conversion from binary code to assembly code, while the depermutator component connects the subroutines of the permutated virus. The normalizer explicitly disregards garbage instructions using the data (Garbage section) from the pattern.

DFA simulation is the final step of the process. Using the input symbol derived from the file being scanned and the automaton created in the building process, the DFA simulator scans the file for malicious content.

For every input symbol, the simulator checks for the matching states and updates them accordingly. Wildcards and conflicts in pattern are resolved by having a set of transition states. When an input symbol is rejected, the DFA simulator checks the entries in the Accept section. If there is a match, the state is toggled back as if it were not rejected. It then reads the next input and continues the simulation. Once the final and accepting state is reached, the file is tagged as a virus.

This particular detection solution covers almost all of the code obfuscation techniques discussed above. Encrypted viruses can be detected by creating a virus signature based on the decryptor's disassembly code. Oligomorphic and polymorphic viruses can be addressed by creating an automaton based on the virus' alphabets or the possible set of instructions that it can produce during infection.

Although polymorphic viruses are capable of producing an almost infinite number of decryptors for each infection, these decryptors can still be subdivided into manageable parts, which enable the creation of a set of automatons. On most occasions, these viruses can be detected generically through detection of the polymorphic engine.

Conveniently, this method also covers the detection of permutating viruses through the depermutator component, which connects the subroutines of the permutated virus.

Unlike emulators, which are known to be slow and cannot handle viruses that generate do-nothing loops, this technique simply treats the virus as a series of disassembly codes that can be matched with a database of existing virus disassembly codes.

For more sophisticated viruses, like Zmist and Etap, this detection method works best if coupled with a smart emulator.

CODE TRANSFORMATION DETECTION

Code transformation is a method of converting mutated instructions to the simplest form where common codes exhibited by the virus can be captured. The combinations of instructions are transformed to an equivalent but simple form.

Etap (aka Simile) is the first metamorphic virus where this type of scanning technology is applicable. Let's have a quick look at how Etap achieves metamorphism through heavy code transformation.

Etap, like Zmist, implements a combination of metamorphic techniques – entry point obfuscation, permutation, and heavy code mutation via shrinking and expanding

techniques. The shrinking and expanding of virus codes is also known as the 'accordion model'. To accomplish such code mutation, this virus had gone through several steps as illustrated below:

Disassembler → Shrinker → Permutator → Expander → Assembler

The virus uses pseudo-assembler techniques to decode each instruction in a form that it can manipulate easily. It extracts the instructions, instruction length, registers and other pertinent information. The shrinker then compresses the disassembled codes produced from the previous generation to prevent the virus code from growing continuously. At this point, garbage codes and do-nothing instructions are also eliminated. Figure 6 shows sample Win32 instructions that Etap has compressed/transformed.

To increase the level of metamorphism, the virus codes are processed first by the permutator. The expander simply undoes what the shrinker did. It replaces the single instructions with corresponding singles, pairs or triplet instructions.

The expander is also responsible for register translation and variable re-selection. Instructions are selected by the expander in a random manner. In the final step, the assembler's task is to convert the pseudo-assembly code into the real Intel IA-32 assembly instructions.

The disassembled code fragments shown in Figures 9 and 10 are two different generations of Etap. At first glance, it

```
XOR Reg,-1          –> NOT Reg
SUB Mem,Imm         –> ADD Mem,-Imm
XOR Reg,0           –> MOV Reg,0
ADD Reg,0           –> NOP
AND Mem,0           –> MOV Mem,0
XOR Reg,Reg         –> MOV Reg,0
SUB Reg,Reg         –> MOV Reg,0
AND Reg,Reg         –> CMP Reg,0
TEST Reg,Reg        –> CMP Reg,0
LEA Reg,[Imm]       –> MOV Reg,Imm
MOV Mem,Mem         –> NOP
```

Figure 6: One-to-one instruction transformation.

```
PUSH Imm / POP Reg           –> MOV Reg,Imm
MOV Mem,Reg/PUSH Me          –> PUSH Reg
MOV Mem,Reg / MOV Reg2,Mem   –> MOV Reg2,Reg
ADD Reg,Imm / ADD Reg,Reg2   –> LEA Reg,[Reg+Reg2+Imm]
OP Reg,Imm / OP Reg,Imm2     –> OP Reg,(Imm OP Imm2)
LEA Reg,[Reg2+Imm] / ADD Reg,Reg3       –> LEA
Reg,[Reg2+Reg3+Imm]
POP Mem / PUSH Mem           –> NOP
MOV Mem2,Mem / MOV Mem3,Mem2 –> MOV Mem3,Mem
OP Reg,xxx / MOV Reg,yyy     –> MOV Reg,yyy
NOT Reg / NEG Reg            –> ADD Reg,1
NEG Reg / ADD Reg,-1         –> NOT Reg
```

Figure 7: Two-to-one instruction transformation.

seems that the two code fragments are different because the instruction constructs used are far from each other. But detailed analysis shows that they both assemble the string 'kernel32.dll' in the stack, then call the GetModuleHandle API.

The solution for Etap is divided into three methods – simple string search, behaviour checking, and code transformation. The first and second method do not guarantee perfect detection and are prone to false positives. The latter is the perfect solution for this type of metamorphism, but is also very hard to implement.

Most anti-virus engines already support string search so it will not be discussed here. The second method requires an emulator to follow the virus code and trigger several flags when a behaviour that pertains to the virus is encountered.

However, this technique does not guarantee perfect detection because there are some samples wherein the API names cannot be resolved properly. In addition, an emulator is required to intercept the RDTSC instruction and make

```
MOV Mem,Reg
OP Mem,Reg2
MOV Reg,Mem          –> OP Reg,Reg2

MOV Mem,Imm
OP Mem,Reg
MOV Reg,Mem          –> OP Reg,Imm (it can't be SUB)

MOV Mem2,Mem
OP Mem2,Imm
MOV Mem,Mem2         –> OP Mem,Imm

CMP Reg,Reg
JNO/JAE/JZ/JBE/JNS/JP/JGE/JLE @xxx
!= Jcc               –> JMP @xxx

MOV Mem,Imm
CMP/TEST Reg,Mem
Jcc @xxx             –> CMP/TEST Reg,Imm
                        Jcc @xxx
                        Jcc @xxx

MOV Mem,Reg
AND/TEST Mem,Reg2
Jcc @xxx             –> TEST Reg,Reg2
                        Jcc @xxx

MOV Mem2,Mem
SUB/CMP Mem2,Reg
Jcc @xxx             –> CMP Mem,Reg
                        Jcc @xxx

MOV Mem2,Mem
AND/TEST Mem2,Imm
Jcc @xxx             –> TEST Mem,Imm
                        Jcc @xxx

MOV Mem2,Mem
SUB/CMP Mem2,Imm
Jcc @xxx             –> CMP Mem,Imm
                        Jcc @xxx

PUSH EAX
PUSH ECX
PUSH EDX             –> APICALL_BEGIN
```

Figure 8: Three-to-one/two/three instruction transformation.

```
mov      eax,06E72656B ;"nrek"
mov      [edx],eax
mov      eax,032336C65 ;"23le"
mov      [edx][04],eax
mov      eax,06C6C642E ;"lld."
mov      [edx][08],eax
xor      eax,eax
mov      edx][0C],eax
call     .00040299D
```

Figure 9: First generation of Etap.

```
push 6c6c442e      ; mov ebp, "lld."
pop ebp
mov edx,73b36c67   ; mov edx, "23le" -> encrypted
and edx,3e7fdedd
push 4e72454b      ; mov esi, "nrek"
pop esi
push ebp           ; mov ecx, ebp
pop ecx
mov dword ptr ds:[42268c],ecx    ; mov mem+8, ecx
lea ebx,dword ptr ds:[esi]       ; mov ebx, esi
mov dword ptr ds:[422684],ebx    ; mov mem, ebx
mov dword ptr ds:[422688],edx    ; mov mem+4, edx
push 0      ; xor reg2, reg2 or mov reg2, 0
pop edx
mov dword ptr ds:[422690],edx    ; mov [mem+c], edx
mov ecx,infect1.00422684         ; mov ecx, mem
push ecx                         ; push ecx
push <&kernel32.getmodulehandlea>      ; mov edi,
offset getmodulehandle
pop edi
call dword ptr ds:[edi]; call getmodulehandle via edi
```

Figure 10: Second generation of Etap.

sure that correct values are specified so that the virus continues with the process. Otherwise, the virus simply exits and the scanner fails to notice the virus behaviour, resulting in a missed detection. Another drawback of this method is that it is slow – because it requires the emulation of every Intel IA-32 instruction.

The last method, which is also the most complex to implement, is code transformation. This method involves transforming the virus code back to its form prior to the expander stage. This form is similar to the first generation as mentioned above.

In this method, we transform the virus code into its simplest form, where common instructions for virus detection are applicable. Three instructions are transformed to two or one instruction(s); two instructions are transformed to one instruction.

The code transformation module must be heavily optimized and flexible to be able to guarantee perfect detection without compromising scanning performance. For speed concerns, the virus location can be transformed from where the scan pattern is taken so as not to cause a significant impact on the performance of the scan engine. Checking filters via geometric techniques like file structure analysis is also advisable, whenever possible.

Zmist uses techniques that are similar to those used by Etap. More details on this virus and its infection techniques can be found in [1].

CONCLUSION

Devising a perfect solution for detection of metamorphic viruses is one of the enduring goals of most anti-virus vendors. Several techniques are known to exist but some are simple workaround solutions while others are shortcuts.

It is important to consider three factors when designing a solution for metamorphic viruses: detection rate, speed and false positives. To achieve perfect detection, a re-write of the scan engine is appropriate to make it interactive, flexible, and optimized to handle such kinds of virus. It is also worth mentioning that creating an automated replication system to produce several different generations of infected samples would be necessary to guarantee a 100% detection rate.

REFERENCES & FURTHER READING

[1] Ször, P. and Ferrie, P.; 'Hunting for Metamorphics', *Proceedings of the Virus Bulletin Conference 2001*, pp. 521–541.

[2] The Mental Driller; 'Metamorphism in practice', *29A*, Issue #6.

[3] Aho, A.V., Sethi, R. and Ullman, J. D.; *Compilers Principles, Techniques, and Tools*, Bell Telephone Laboratories 1986.

[4] Sedgewick, R.; *Algorithms (Second Edition)*. Addison-Wesley 1986.

[5] Qozah, 'Polymorphism and Grammars', *29A* Issue #4, 1999.

[6] Tanenbaum, A.S.; *Structured Computer Organization*, Prentice-Hall 1999.

[7] Grune, R. and Jacobs, C.J.H.; *Parsing Technique – A Practical Guide,* Amstelveen/Amsterdam, 1990.

[8] Myers, E.W.; Oliva, P.; and Guimaraes, K.; 'Reporting Exact and Approximate Regular Matches', 1988.

[9] Wu, S. and Manber, U.; 'Fast Text Searching with Errors', University of Arizona.

[10] Perriot, F. and Ferrie, P.; 'Principles of X-Raying', *Proceedings of the Virus Bulletin Conference 2004*, pp. 51–66.

END NOTES & NEWS

The 9th annual WEBSEC conference takes place 27–31 March 2006 in London, UK. The event will include live hacking demos, a network and application hacker challenge and more than 40 sessions on topical security issues including a panel debate in which *Virus Bulletin*'s Technical Consultant Matthew Ham will be a panel member. For more details see http://www.mistieurope.com/.

The 2006 e-Crime Congress takes place 30–31 March 2006 in London, UK. The Congress will seek to challenge conventional attitudes on e-crime and examine how business, government and law enforcement can continue to work together to tackle threats that undermine public confidence in the Internet. For details see http://www.e-crimecongress.org/.

The 2nd Information Security Practice and Experience Conference (ISPEC 2006) will be held 11–14 April 2006 in Hangzhou, China. For details see http://ispec2006.i2r.a-star.edu.sg/.

Infosecurity Europe 2006 takes place 25–27 April 2006 in London, UK. For details or to register interest in the event see http://www.infosec.co.uk/.

The 15th EICAR conference will take place from 29 April to 2 May 2006 in Hamburg, Germany. Authors are invited to submit posters for the conference. The deadlines for submitting poster presentations is 24 February 2006. For more information see http://conference.eicar.org/2006/.

The Seventh National Information Security Conference (NISC 7) will take place from 17–19 May 2006 at St. Andrews Bay Golf Resort & Spa, Scotland. Enquiries may be directed to tina.deighton@sapphire.net or via http://www.nisc.org.uk/ .

The 2006 IEEE Symposium on Security and Privacy will be held 21–24 May 2006 in Oakland, CA, USA. For details see http://www.ieee-security.org/TC/SP2006/oakland06.html.

AusCERT 2006 takes place 21–25 May 2006 in Gold Coast, Australia. A programme overview, providing a list of confirmed speakers, can be found at http://conference.auscert.org.au/.

The Fourth International Workshop on Security in Information Systems, WOSIS-2006, will be held 23–24 May 2006 in Paphos, Cyprus. For details see http://www.iceis.org/.

CSI NetSec '06 takes place 12–14 June 2006 in Scottsdale, AZ, USA. Topics to be covered at the event include: wireless, remote access, attacks and countermeasures, intrusion prevention, forensics and current trends. For more details see http://www.gocsi.com/.

Black Hat USA 2006 will be held 29 July to 3 August 2006 in Las Vegas, NV, USA. Online registration for the event will be available from 15 March. See http://www.blackhat.com/.

The 15th USENIX Security Symposium takes place 31 July – 4 August 2006 in Vancouver, B.C., Canada. A training programme will be followed by a technical programme, which will include refereed papers, invited talks, work-in-progress reports, panel discussions and birds-of-a-feather sessions. A workshop, entitled Hot Topics in Security (HotSec '06), will also be held in conjunction with the main conference. For more details see http://www.usenix.org/.

HITBSecConf2006 will take place 16–19 September 2006 in Kuala Lumpur. Further details and a call for papers will be announced in due course at http://www.hackinthebox.org/.

Black Hat Japan 2006 takes place 5–6 October 2006 in Tokyo, Japan. Unlike other Black Hat events, Black Hat Japan features Briefings only. For more information see http://www.blackhat.com/.

The 16th Virus Bulletin International Conference, VB2006, will take place 11–13 October 2006 in Montréal, Canada. For details of sponsorship opportunities, please email vb2006@virusbtn.com. Online registration and full programme details will be available soon at http://www.virusbtn.com/.

RSA Conference Europe 2006 takes place 23–25 October 2006 in Nice, France. See http://2006.rsaconference.com/europe/.

AVAR 2006 will be held 4–5 December 2006 in Auckland, New Zealand. More details will be announced in due course at http://www.aavar.org/.

SUBSCRIPTION RATES

Subscription price for 1 year (12 issues):

- Single user: $175
- Corporate (turnover < $10 million): $500
- Corporate (turnover < $100 million): $1,000
- Corporate (turnover > $100 million): $2,000

Corporate rates include a licence for intranet publication.

See http://www.virusbtn.com/virusbulletin/subscriptions/ for subscription terms and conditions.

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 Web: http://www.virusbtn.com/

vb Spam supplement

CONTENTS

NEWS & EVENTS

DUTCH POLICE ARREST NIGERIAN SCAMMERS

Dutch police arrested 12 Nigerians in Amsterdam last month after they were found to be operating a 419 scamming ring. According to Dutch police, who were aided in their investigation by members of the US Postal Inspection Service, the scammers had sent more than 100,000 emails and had managed to con gullible victims – mainly from the US – into handing over a total of more than $2 million.

Four of the Nigerians will be extradited to face charges in the US, while the others will face charges in the Netherlands.

CHINA TO CRACK DOWN ON SPAM

The Chinese Government has introduced a set of regulations aimed at reducing the amount of spam circulating in the country. The sending of advertising emails without the recipient's prior permission is now banned, and all advertising emails are required to be labelled as such. Spammers will now face penalties of up to 30,000 yuan (US$3,750), and an offence-reporting centre (www.anti-spam.cn) has been launched, at which users can register their complaints about being spammed.

According to the country's Ministry of Information Industry (MII), approximately 60 per cent of the 50 billion emails sent and received in China last year were spam. Furthermore, the hitherto slack regulation of junk mail in the country has attracted spammers from across the globe to use servers based in China to send out their spam.

According to statistics reported by both *Sophos* and *CommTouch*, China currently has the dubious honour of being the second-largest spam-producing country in the world (being 'out-spammed' only by the United States) – it is hoped that the new regulations will go some way towards reducing the problem.

VIGILANT STAFF AVERT PHISHING SCAM

A web-hosting company based in New Zealand claims that, thanks to the vigilance of its staff, it has averted a potential phishing scam targeted at customers of *National Bank*.

Staff of Auckland-based *Web Drive* suspected that something fishy (or phishy) was going on when they spotted an overseas user who had registered around 25 variations of the *National Bank*'s web address using several different credit card numbers.

Web Drive staff were quick to react, cancelling the domains and re-registering them in the web hosting company's own name to prevent them from being registered elsewhere.

EVENTS

The 2006 Spam Conference will be held 28 March 2006 at MIT, Cambridge, MA, USA. For details see http://www.spamconference.org/.

The Authentication Summit II takes place on 19 April 2006 in Chicago, IL, USA. The conference will cover the latest advances in email authentication, including Sender ID Framework (SIDF) and DomainKeys Identified Mail (DKIM), with a focus on real-life results and prescriptive information. For full details see http://emailauthentication.org/.

INBOX 2006 will be held 31 May to 1 June 2006 in San Jose, CA, USA. The event will cover all aspects of email including topics such as 'has CAN-SPAM failed us?', 'what can ISPs do to fix spam?', 'how not to be a spammer' and 'new directions in identifying spam'. For more information see http://www.inboxevent.com/2006/.

The third Conference on Email and Anti-Spam, CEAS 2006, will be held 27–28 July 2006 in Mountain View, CA, USA. Those wishing to present papers are invited to submit their proposals before 23 March 2006. Full details can be found at http://www.ceas.cc/.

The Text Retrieval Conference (TREC) 2006 will be held 14–17 November 2006 at NIST in Gaithersburg, MD, USA. More information about the TREC 2006 spam track can be found at: http://plg.uwaterloo.ca/~gvcormac/spam/.

FEATURE

FIGHTING SPAM WITH DATA COMPRESSION MODELS

Andrej Bratko
Jozef Stefan Institute, Slovenia

We know from experience that emails advertising prescription drugs are typically spam, so an email containing the word 'viagra' is usually destined straight for the junk folder. But there are reportedly 1,300,925,111,156,286,160,896 ways to spell 'viagra', all of which are legible to the human eye [1]. How can we train an automatic filter to cope with this diversity of natural language?

Although a spam filter might anticipate all plausible and implausible character substitutions and spelling variations, spammers have a reputation for inventing new, creative ways of getting their message across. To complicate matters further, formatted text provides infinite possibilities for additional obfuscation.

Surely, there must be a more elegant solution.

In the following article we explore a filtering approach that is less sensitive to such obfuscation tactics, by treating email as a sequence of characters, rather than a collection of words.

THE TOKENIZATION PROBLEM

Most content-based spam filters convert the textual contents of an email into a set of attributes that describe the message in machine readable form. This is commonly referred to as the 'bag-of-words' representation, reflecting the fact that attributes generally correspond to word-like tokens extracted from the text. Statistical filters use this representation in combination with some machine learning algorithm.

At best, these algorithms can learn to discriminate spam from legitimate email in the token world. Their success depends crucially on the assumption that the token world adequately approximates the actual information conveyed in the messages, as interpreted by a human. However, this is not always the case, especially in the adversarial spam filtering setting.

An obvious caveat here is that there is no *a priori* definition of similarity between tokens. The words 'viagra', 'vi@gra' and 'v1agra' are simply considered different and the filter is unable to generalize whatever it has learned about one to the other.

It is this weakness that is commonly exploited by spammers with various obfuscation tactics, such as common character substitutions and formatting tricks. Although the success of such tactics is hard to measure in practice, a good indication of their effect is their widespread use in the wild. It is clear that spammers would not obfuscate messages if these tactics were fruitless.

Many spam filters take measures to detect obfuscation, or even try to deobfuscate messages. We explore a different route here by omitting the tokenization step altogether and using data compression algorithms that operate directly on character sequences.

STATISTICAL DATA COMPRESSION MODELS

Probability plays a central role in data compression. The general idea is to assign shorter codes to more probable characters and longer codes to less probable ones. In fact, if we know the exact probability distribution over all possible messages emitted by some information source, we can compress these messages optimally. Here, messages are independent sequences of characters, which can be arbitrarily long. Given some training data, statistical data compression algorithms learn a probability distribution over such sequences.

Compression algorithms learn, for example, that the text 'The weather is pleasant today' is much more probable than 'Vreme je danes prijetno' in English text. However, had we trained the algorithm on a Slovenian text corpus, the latter sequence would undoubtedly receive a greater probability.

It is straightforward to see how these algorithms can be used to determine the language of an unknown piece of text, or, along the same lines, to determine whether a piece of text is in fact a piece of spam. We simply train two compression models, one from examples of spam, and another from examples of ham (legitimate email). The classification outcome is determined by the model that finds the target text more likely.

Formally, Bayes rule is used for classification by approximating the conditional document probability given the class, with the probability assigned to the document by the respective compression model. The theoretical relationship between data compression and Bayesian classification is elaborated further in our technical report, which is available online [2].

ESTIMATING THE PROBABILITY OF A SEQUENCE OF CHARACTERS

So how exactly do data compression methods estimate the probability of every possible sequence of characters? They

certainly cannot be estimated directly, since there are an infinite number of such sequences.

To make the inference problem tractable, sources are usually modelled as Markov sources with limited memory. This means that the probability of each character in a sequence depends only on its context – a limited number of characters immediately preceding it. The probability of a sequence of n characters $s_1^n = s_1 \ldots s_n$ is then

$$P(s_1^n) = \prod_{i=1}^{n} P(s_i \mid s_{i-k}^{i-1})$$

The number of context characters, k, that are used for prediction is usually referred to as the order of the compression model. The parameters of such models are next-character probabilities, and a different set must be learned for each possible combination of preceding characters.

Higher order models are more realistic – however such models also require a lot of training data to obtain accurate parameter estimates, since the number of parameters grows exponentially with the order of the model. For example, an order-k model requires $256^k \cdot 255$ parameters, if we assume that each character has 256 possible values.

Compression algorithms generally try to make the most of the limited training data available by blending the predictions of different order models. The idea is to use higher-order models only when the amount of data is sufficient to support them.

The next section contains a fairly condensed technical description of the PPM compression model, and can be skipped on first reading.

THE PPM COMPRESSION ALGORITHM

The Prediction by Partial Matching (PPM) compression algorithm is among the best-performing compression schemes for lossless text compression.

An order-k PPM model works as follows: when predicting the next character in a sequence, the longest context found in the training text is used, up to length k. If the target character has appeared in this context in the training text, its relative frequency within the context is used as an estimate of the character's probability.

These probabilities are discounted to reserve some probability mass, which is called the escape probability. The escape probability is effectively distributed among characters not seen in the current context, according to a lower-order model.

The procedure is applied recursively until all characters receive a non-zero probability, ultimately terminating in a default model of order –1, which always predicts a uniform distribution among all possible characters.

Many versions of the PPM algorithm exist, differing mainly in the way the escape probability is estimated. In our implementation, we used what is known as escape method D, which simply discounts the frequency of each observed character by 1/2 occurrence and uses the gained probability mass as the escape probability.

HOW WELL DOES IT PERFORM?

A spam filter based on the PPM-D compression algorithm was submitted for evaluation in the spam filtering track of the 2005 Text REtrieval Conference (TREC). The goal of the spam track is to provide a standard evaluation of current and proposed spam filtering approaches. (For an overview of the TREC 2005 spam track, including a description of the evaluation tools and measures, see *VB*, January 2006, p.S2.)

The four corpora used for the evaluation contained a total of 318,482 messages, 113,129 of which were spam.

An online learning scheme that mimics real email usage was adopted for the evaluation: messages are presented to the filter in chronological order. For each message, the spam filter must first make a guess as to how likely it is that the message is spam, after which it is communicated the gold standard judgment representing the true classification of the message as spam or ham. This allows the filter to update its statistics before assessing the next message.

The performance of spam filters is usually measured in spam misclassification and false positive rates. There is typically a trade off between these two measures, so that one of them can be improved at the expense of the other by adjusting a fixed filtering threshold. The higher the threshold, the less likely we are to falsely identify a

Filter	ROCA%	SMR% at 0.1% FP	SMR% at 0.01% FP
ijsSPAM2	0.051	3.78	19.81
crmSPAM2	0.115	3.46	29.30
lbSPAM2	0.132	6.75	25.84
tamSPAM1	0.172	9.10	77.77
yorSPAM2	0.316	21.14	45.69
kidSPAM1	0.768	66.13	96.34
621SPAM1	1.008	4.36	40.67

Table 1: Comparison of filters in the area above the ROC curve statistic (ROCA) and spam misclassification rates (SMR) at 0.1% and 0.01% false positives. A smaller number indicates a better performance.

Figure 1: How does PPM see messages? Each character is coloured according to how 'spammy' it looks to the filter: red indicates spam, green indicates ham (good email).

legitimate message as spam, but more spam will also make it past the filter.

The primary measure used for evaluating spam filters in TREC was the area above the Receiver Operating Characteristic curve (ROCA). This measure equals the probability that a legitimate message appears more 'spammy' to the filter than a spam message (a certain error), which is estimated over all pairs of legitimate and spam messages in the dataset. The measure captures the filter's behaviour across all possible filtering thresholds.

Table 1 lists summary performance measures on the aggregate results over all four corpora used for TREC. Only the seven best-performing systems from different organizations are shown here – a comprehensive list of all results can be found in the TREC proceedings [3].

The table lists the filters in order of decreasing performance according to the area above the ROC curve statistic. The list includes open source filters, such as *CRM114* (crmSPAM2), *DBACL* (lbSPAM2) and *Spam-Bayes* (tamSPAM1), as well as *IBM*'s *SpamGuru* (621SPAM1). The system developed at the Jozef Stefan Institute (labelled ijsSPAM2) is based on an order-6 PPM-D compression model.

WHAT ARE THE TELL-TALE FEATURES?

The primary motivation for using data compression as a spam filtering tool is that the approach conveniently circumvents the need for tokenization. Patterns of punctuation and other special characters which are hard to model as a 'bag-of-words' are naturally included in the model.

Figure 1 shows some examples of the types of patterns picked up by the PPM model – each character is coloured according to how 'spammy' it looks to the filter, with red indicating spam and green indicating ham. Typical character substitutions used by spammers are recognized easily, as are URL obfuscation tactics. Useful patterns can also be found in the non-textual message headers.

Some interesting patterns that are typical of legitimate email are also shown, such as message headers that are only included in replies and line prefix characters that usually accompany forwarded text.

CONCLUSION

Character-based compression models are a promising technique to help future statistical spam filters combat the ever-increasing volumes of spam as well as newer, more malicious forms of unsolicited email. The mere fact that these algorithms operate in the world of character sequences makes it more difficult to find ways of defeating them. Surely, any form of obfuscation only makes it easier for the filter to recognize that the message is spam.

Although no publicly available spam filters currently use data compression algorithms, any off-the-shelf implementation of PPM, such as Charles Bloom's PPMZ [4], can easily be adapted for this task.

REFERENCES

[1] 'How many different ways are there to spell Viagra?'; http://cockeyed.com/lessons/viagra/viagra.html.

[2] Bratko, A. and Filipic, B.; 'Spam filtering using compression models'; http://ai.ijs.si/andrej/papers/ijs_dp9227.html.

[3] Proceedings of the 14th Text REtrieval Conference (TREC 2005); http://trec.nist.gov/pubs.html.

[4] Charles Bloom's PPMZ; http://www.cbloom.com/src/ppmz.html.