WHITE PAPER: SYMANTEC SECURITY RESPONSE

Me code write good: The l33t skillz of the virus writer.

John Canavan
Symantec Security Response, Dublin

Me code write good: The l33t skillz of the virus writer.

Contents

Abstract

Viruses and worms pose some of the most formidable threats in the modern computer security landscape. With some virus writers on the bleeding edge of technology, making use of 0-day exploits and innovative techniques to circumvent system security features.

However, for every Blaster, there's a worm that repeatedly attempts to infect the same machine. For every 100,000 node botnet Spybot infection there's 20 variants that fail to get as far as even connecting to an IRC server. For every Netsky, there's an intended mass mailer that crashes before it sends a single copy of itself out.

From exploitable vulnerabilities in their code to incomprehensible goofs there's no shortage of evidence that a large proportion of virus writers aren't quite as capable as they would like others to think. This paper will take a look at the legacy of these slightly less than expert level virus writers, and examine the threat they continue to pose.

Introduction

Bugs in software code are nothing new. From the early days of computing, bugs have been found in programs from all areas of technology.

In July 1962 a bug in the flight software for the Mariner 1 space probe caused the rocket to divert from its intended path after launch. Veering off course and heading for a crash in the North Atlantic shipping lanes, the Range Safety Officer destroys the rocket. Afterwards, an investigation discovered an error had occurred when an equation was being transcribed by hand in the specification for the guidance program.[1]

A race condition in the controlling software for the Therac-25 radiation therapy machine was responsible for giving patients massive overdoses of radiation. Between 1985 and 1987 at least 5 patients died as a direct cause. Amongst other things, there was no check in place to prevent the electron-beam from operating in its high-energy mode without the target in place.[2]

In August 2003, a massive power blackout affected the north-eastern coast of the United States. Fearing a terrorist attack, many fled cities in search of safety. What actually happened? It was discovered that a bug in the energy management software used by the utility company was triggered by a unique combination of events and alarm conditions on the equipment it was monitoring. When a backup server kicked in, it also failed, unable to handle the accumulation of unprocessed events that had queued up since the main system's failure. Because the system failed silently, energy company operators were unaware that they were looking at outdated information on the status of their portion of the power grid for over an hour, according to the report from the U.S.-Canadian task force investigating the blackout[3]. The power outage that followed plunged the millions into darkness and caused a significant degree of panic, receiving worldwide blanket media coverage.

Bugs in Viruses

Given that these major errors can occur in even the most thoroughly tested professionally written code, it's not surprising that computer viruses have their fair share of malfunctions and buggy code.

Typically written by amateur fanatics, hacked together by script kiddies or as a form of experimentation by overly-curious fledgling coders, viruses rarely undergo any sort of stringent testing before their release into the wild. It is not uncommon to find threats with bugs in their code, indeed some of the most high profile and prevalent threats in recent years have had bugs.

And so with that in mind we can take a look at some case studies of bugs in viruses, from simple mathematical errors, to those that cause key functionality to misfire. We will analyze how they affected the spread of the virus, what effect, if any, they had on the payload, and most importantly what impact they had on infected systems and networks. We will examine what lessons can be learned by the virus analysts in looking back at these threats; areas where bugs are common; how analysts can pinpoint them in a more timely manner and get that information back to their customers as soon as possible.

In each example, we will take a brief look at the background of the virus, and the intentions of the virus writer by way of a short **analysis** of the threat's main functions. We will then study the key **bugs** in the code that hamper functionality, and take note of how these bugs affected the **impact** the threat had in the wild.

The Morris Worm

Having examined two of the more recent and high-profile threats affected by buggy code, we will now take a look at what is widely renowned as the first blended threat Internet worm.

On the evening of November 2nd, 1988 the Internet came under attack from the Morris Worm. Taking advantage of flaws in *fingerd/gets* and *sendmail* in BSD-derived versions of UNIX, the worm spread quickly, causing confusion and consternation of system administrators and users as they discovered that their systems had been compromised. This frustration quickly grew as systems became overloaded with running processes as they became repeatedly infected. As time went on, many machines surrendered to the crippling load and failed completely as their swap space or process tables were exhausted.[4]

As the first threat of its kind, response was initially slow, but within 12 hours the Research Group at Berkeley had developed a temporary solution to halt its spread. Patches for the vulnerabilities that the worm exploited as an infection vector were posted the next day, and the situation was under control.

Analysis

Allegedly written to gauge the size of the Internet, the Morris Worm was simply designed to spread to as many systems as possible, and to that end it used three different attack vectors: *rsh*, *fingerd* and *sendmail*.

- The worm first attempted to spawn a remote shell, invoking /usr/ucb/rsh, /usr/bin/rsh, and /bin/rsh.
- If this failed the worm connected to the remote finger server daemon sending a 536 byte buffer overflow exploit string to execute an execve("/bin/sh",0, 0). This attack only worked on vulnerable VAX machines, and caused a core dump on Suns.
- Finally the worm would attempt to exploit an SMTP vulnerability, setting debug on and sending:

```
mail from: </dev/null>
rcpt to: <"|sed -e '1,/^$/' d | /bin/sh ; exit 0">
```

The worm consisted of two parts, the main viral body, and the bootstrap/vector program. Once the remote system was exploited, the worm sent the vector program, which was then executed. The vector program connected back to the host machine, and downloaded the other worm components. Once the main viral body of the worm was executed, it gathered some local system information, iterated through the password file attempting to crack local user passwords, then began to look for more targets to infect.

Bugs

While the Morris Worm did manage to infect a large number of systems in a relatively short time period, it is littered with bugs and poor coding practice. Calls are made to functions with incorrect numbers of arguments, local variables are declared but never used, and the code even includes routines that are never referenced, and others that will not be executed because of conditions that are never met.

```
   if ((random() % 7) == 3)
return;
```

In an attempt to prevent infected systems grinding to a halt while being overrun with worm processes, functionality was included to thin out local worm processes. With a probability of 1 in 7, as above, the routine returned without doing anything, in these cases the worm never returned to this function to recheck for local infections. Otherwise the worm connected to local port 23357 to check for an infection with an exchange of challenges.

After the client exchanges magic numbers with the server as a trivial form of authentication, the client and the server roll dice to see who gets to survive. If the exclusive-or of the respective low bits of the client's and the server's random numbers is 1, the server wins, otherwise the client wins. The

loser sets a flag *pleasequit* that eventually allows it to exit at the bottom of the main loop. If at any time a problem occurs – a read from the server fails, or the wrong magic number is returned – the client worm returns from the function, becoming a worm that never acts as a server and hence does not engage in population control, becoming immortal.[5]

There were a number of situations that this routine didn't properly account for. For example, if:

- Several viruses infected a clean machine at once, in which case all of them would look for listeners; none of them would find any; all of them would attempt to become listeners; one would succeed; the others would fail, give up, and thus be invulnerable to future checking attempts.
- Several viruses start at once, in the presence of a running virus. If the first one "wins the coin toss" with the listening virus, other commencing versions will have contacted the losing one and have the connection closed upon them, permitting them to continue.
- A machine is slow or heavily loaded, which could cause the virus to exceed the timeouts imposed on the exchange of numbers, especially if the compiler was running (possibly multiple times) due to a new infection. Note that this is exacerbated by a busy machine (which slows down further) on a moderately sized network.

Impact

The most noticeable result of the design flaws and bugs in the Morris Worm was that it propagated far faster than expected, and with many instances of the worm draining resources on single systems, it was caught very quickly.

This allowed experts to move early on getting removal instructions to system administrators, and begin work on patches immediately.

The spread of this virus was to be effective in raising the awareness of users (and administrators) to the importance of choosing 'difficult' passwords. Tools duplicating the worm's password attack were made available in order to allow administrators to analyze the passwords in use on their systems.

The program could have been much more virulent had the author been more experienced or less rushed in her/his coding. However, it seems likely that this code had been developed over a long period of time, so the only conclusion that can be drawn is that the author was sloppy or careless (or both), and perhaps that the release of the worm was premature.

Sobig, Sobad

The Sobig family was a series of mass-mailing worms, the first of which emerged in early January 2003, with a total of six variants released over the following eight months. Named for the large size of its code, in comparison to similar mass-mailers of the time, its initial purpose was to spread a proxy

server Trojan, which the author had put together the previous year.[6]

Setting records for the speed of their propagation, further variants were released to increase the number of these proxies accessible for the sending of spam. It was estimated in reports of the time that infections numbered in the millions, and reports of impending new date-triggered functionality were widespread.[7]

Over the six variants, the core base-code changed very little, so we will look at a brief overview of how they worked, and then take examples from several variants to see some mistakes that were made.

Analysis

A typical mass mailer, Sobig sends itself to all the addresses it finds in the .txt, .eml, .html, .htm, .dbx, and .wab files. Using spoofed *From* addresses, the worm had Subjects such as: “Re: Movies”, “Re: Here is that sample” and “Re: Wicked screensaver”, asking users to “See the attached file for details”.

After copying itself to the %System% folder, it attempts to copy itself to accessible network shares, and then proceed to mail itself out.

Bugs

In what may well be a record, Sobig opened their first line of code with a bug. The worm’s random number generator is initialized, using the current time as the seed. However this initialization is only done once, in the main thread, whereas the worm uses multiple threads using Thread Local Storage. Since the random number generator is not initialized in those other threads (seed begins at zero), it will result in the same sequence each time the worm is executed.

The functionality of each Sobig variant is date-triggered, with the worm restricted to operating between certain dates. After initializing its random number generator the next thing checked is that the current date is within the specified date range. Sobig.A converts the current date to *yyyy.mm.d* format, and compares it against 2003.1.23. However, as mm is specified in the format, Windows returns the month in two-digit form adding a leading 0 to months before October (i.e. the 10th month), so this comparison always fails. Later variants of the worm solve this problem with a new date check function.

In what appears to be an attempt to prevent multiple copies of the worm executing at the same time, Sobig uses a named event. When a named event is created, it’s name is added to the global namespace of Windows, which means a *CreateEvent()* call, as seen at location *409521* of the code extract below, will not return a failure for an event that exists already. This leads to a race condition that can allow several copies of the worm to run at the same time.

We can also see in the extract below at location *409567*, that the worm initializes Winsock support, looking for version 2.2, but ignores the version that is returned.

```
.shrink:00409517                 push    ds:lpazTrayX    ; lpName
.shrink:0040951D                 push    esi             ; bInitialState
.shrink:0040951E                 push    1               ; bManualReset
.shrink:00409520                 push    esi             ; lpEventAttributes
.shrink:00409521                 call    ds:CreateEventA
.shrink:00409527                 mov     edi, ds:WaitForSingleObject
.shrink:0040952D                 push    esi             ; dwMilliseconds
.shrink:0040952E                 push    eax             ; hHandle
.shrink:0040952F                 mov     [ebp+hOwnEvent], eax
.shrink:00409532                 call    edi ; WaitForSingleObject
.shrink:00409534                 test    eax, eax
.shrink:00409536                 jnz     short ok_1
.shrink:00409538                 or      esi, 0FFFFFFFFh
.shrink:0040953B                 jmp     done
.shrink:00409540 ; ---------------------------------------------------------------------
.shrink:00409540
.shrink:00409540 ok_1:                                   ; CODE XREF: WinMain(x,x,x,x)+AA_j
.shrink:00409540                 push    [ebp+lpData]    ; char *
.shrink:00409543                 call    _strlen
.shrink:00409548                 test    eax, eax
.shrink:0040954A                 pop     ecx
.shrink:0040954B                 jz      no_commandline
.shrink:00409551                 mov     ebx, [ebp+hOwnEvent]
.shrink:00409554                 push    ebx             ; hEvent
.shrink:00409555                 call    ds:SetEvent
.shrink:0040955B                 lea     eax, [ebp+CommandLine]
.shrink:00409561                 push    eax             ; lpWSAData
.shrink:00409562                 push    202h            ; wVersionRequested
.shrink:00409567                 call    WSAStartup
.shrink:0040956C                 test    eax, eax
.shrink:0040956E                 jz      short ok_2
```

When constructing the email to send out, the worm opens the copy of itself to attach by calling CreateFileA. However it does so with ShareMode set to zero. With multiple threads for sending emails, this can result in a failure to open the file. This isn't checked and the email is sent anyway, without an attachment.

In an attempt to infect available network shares, Sobig.F takes a filename from those found on shares while searching for email addresses, and appends .exe to it, creating an executable filename as a destination to copy itself to. After this filename is chosen, nothing happens. The code to copy the file over is omitted.

Impact

Lots of small bugs again indicate a lack of proper testing, but still do not impede the spread of the virus.

Even though it uses no exploits to propagate, and its network share code does not work, leaving it to rely entirely on the minimal social engineering tricks of its emails, Sobig.F still became one of the biggest viral threats in 2003.

Kama Sutra a wet blanket

In early 2006, W32.Blackmal.E@mm (Nyxem,MyWife) and OSX.Leap both created a media frenzy, only to fizzle out quickly thereafter due to elementary bugs in their code.

W32.Blackmal.E@mm surfaced in late January, mass-mailing itself from infected hosts with a choice of 19 mail subjects. Blackmal.E had the novel feature of contacting a web-based script which functioned as an infection counter. Within days of its emergence this counter had reached over half a million. Some estimated as many as 1.8 million infected machines.[8]

Analysis

As well as crippling Anti-Virus and security products installed on the infected machine, by deleting key files and registry entries, Blackmal.E included a routine, activated on the 3rd of the month, designed to overwrite files with the following extensions in available drives A through Z:

*.doc *.xls *.mdb *.mde

*.ppt *.pps *.zip *.rar

*.pdf *.psd *.dmp

With the following text:

```
DATA Error [ 47 0F 94 93 F4 F5]
```

Bug

Written in VB the following extract is decompiled to VBS pcode.

```
L4: On Error GoTo Next
L7: On Error Resume at $+46
L9: push &(Variant[ -0054]  = 1 As Integer)
L14: push &[ -001C]
L17: push &[ -0022]
L20: With currentobject=this
L21: call [ currentobject.method 203], push Long result
L24: Object[ -0020]  = pop (no addref on source)
L27: With currentobject=[ -0020]
L30: call [ currentobject.method 44]  117, check result
```

The worm creates a VB Drive Control object, which contains links to all available drives on the infected system. However in attempting to enumerate the drives we see on the 3rd line above, the worm initializes its index at 1, thus skipping the first drive on the system.

Impact

However, when D-Day arrived on February 3, the widely anticipated reports of chaos and destruction across the world never materialized. Why? The media coverage that the threat received certainly raised awareness of the threat high enough so that many people had protected themselves. However something that was overlooked in a lot of reports at the time was this bug in the code, which meant that the worm would not overwrite files on the first available drive found. For example if the first available drive is the C drive, the worm will overwrite files in available drives from D to Z.

This would have seen a significant number of newer systems store their data safely from the destructive payload of the worm.

Osx.leap jumps the gun

Shortly after the storm surrounding Blackmal died down, new reports emerged that Mac OS X was under threat from a new instant messaging worm/virus/trojan.

- First Mac OS Trojan[9]
- First ever virus for Mac OS X discovered[10]
- Mac users face first Apple virus[11]
- First Mac OS X Malware Infects Via iChat[12]

While it was not the first appearance of malicious code on Mac OS X, the incident received major attention from the Apple community, and the technology media at large.

Originally posted on a “hacking” web forum purporting to be pictures of MacBook Pro Internals, it was then reposted on macrumours.com claiming to be screenshots of "MacOS X Leopard" (an upcoming version of MacOS X, aka "MacOS X 10.5"). The worm uses social engineering tactics to attempt to propagate via iChat. However, even once the file has been received, in order to become infected the user still needs to decompress the archive, and execute the contained file.

Analysis

When executed, the worm copies a number of files to /tmp – mostly various copies of itself for later use. It then creates an Input Manager called apphook.bundle in /Library/InputManagers/ (for root users, or ~/Library/InputManagers/ otherwise), deleting any existing “apphook” bundles in that folder. Now, when any application is launched, the OS loads the newly installed “apphook” Input Manager into its address space.

Once this hook is in place, the worm then searches for and infects the four most recently used applications this month by copying the contents of the data fork to the resource fork of the selected file, and then copying itself to the data fork of the selected file. The worm is then executed any time the infected application is run. When launched from an infected application the worm calls an execv on the resource fork of the executable, intending to launch the original application as would normally happen.

When iChat is launched the worm will attempt to send a copy of itself, as latestpics.tgz, to all contacts in the local Bonjour!/Rendezvous buddy list.

Bugs

```
__text:00002D68 38 63 00 04 addi %r3, %r3, 4 # size_t
__text:00002D6C 48 00 09 B5 bl _malloc_stub
__text:00002D70 7C 7A 1B 79 mr. %r26, %r3
__text:00002D74 41 82 00 68 beq loc_2DDC
__text:00002D78 80 7E 00 00 lwz %r3, 0(%r30) # char *
__text:00002D7C 48 00 0A 45 bl _strlen_stub
__text:00002D80 38 80 00 69 li %r4, 0x69
__text:00002D84 3B 63 00 04 addi %r27, %r3, 4
__text:00002D88 3C 7F 00 00 addis %r3, %r31, 0
__text:00002D8C 38 63 0F 88 addi %r3, %r3, (aSS - loc_2CF4) # "%s%s"
__text:00002D90 4B FF FE F1 bl _xor
__text:00002D94 83 9E 00 00 lwz %r28, 0(%r30)
__text:00002D98 7C 7D 1B 78 mr %r29, %r3
__text:00002D9C 3C 7F 00 00 addis %r3, %r31, 0
__text:00002DA0 38 80 00 69 li %r4, 0x69
__text:00002DA4 38 63 0F 90 addi %r3, %r3, (a__namedforkRsr - loc_2CF4) # "/..namedfork/rsrc"
__text:00002DA8 4B FF FE D9 bl _xor
__text:00002DAC 7F 64 DB 78 mr %r4, %r27
__text:00002DB0 7C 67 1B 78 mr %r7, %r3
__text:00002DB4 7F A5 EB 78 mr %r5, %r29
__text:00002DB8 7F 43 D3 78 mr %r3, %r26
__text:00002DBC 7F 86 E3 78 mr %r6, %r28
__text:00002DC0 48 00 05 F1 bl _snprintf_LDBLStub
__text:00002DC4 7F 43 D3 78 mr %r3, %r26 # char *
__text:00002DC8 7F C4 F3 78 mr %r4, %r30 # char **
__text:00002DCC 7F 25 CB 78 mr %r5, %r25 # char **

__text:00002DD0 48 00 09 B1 bl _execve_stub
```

Above is code from OSX.Leap that is intended to execute the original application now contained in the resource fork of the infected file. The *malloc* call on the 2nd line allocates memory to be used to append the string "/..namedfork/rsrc" to the path. However, as we can see in the first line, instead of adding the length of the string, it errantly adds the length of the *pointer* to the string, which is always 4 bytes. This means that the *snprintf* call will result in only "/.." and a trailing terminating null character to be appended to the string.

This bug means that any applications infected by the virus will not be executed.

Osx.Leap.A also contains a bug in its iChat code that may corrupt the file so that it appears larger than it actually is. In some cases, this can result in the file failing to be sent successfully.

Impact

Being so heavily reliant on social engineering to infect a system was always going to restrict OSX.Leap.A's ability to propagate in the wild. The fact that it only uses iChat contacts from the local Bonjour!/Rendevous list reduces it's targets significantly. And with bugs in its iChat code that reduce the effective rate of transfer even further, the chances of OSX.Leap.A infection was low.

However, bugs in the file infection routine, as illustrated above, mean that once infected, the payload is significantly more damaging than intended. Each time the threat is run, your four most recently used applications are infected, and with that infection, rendered useless.

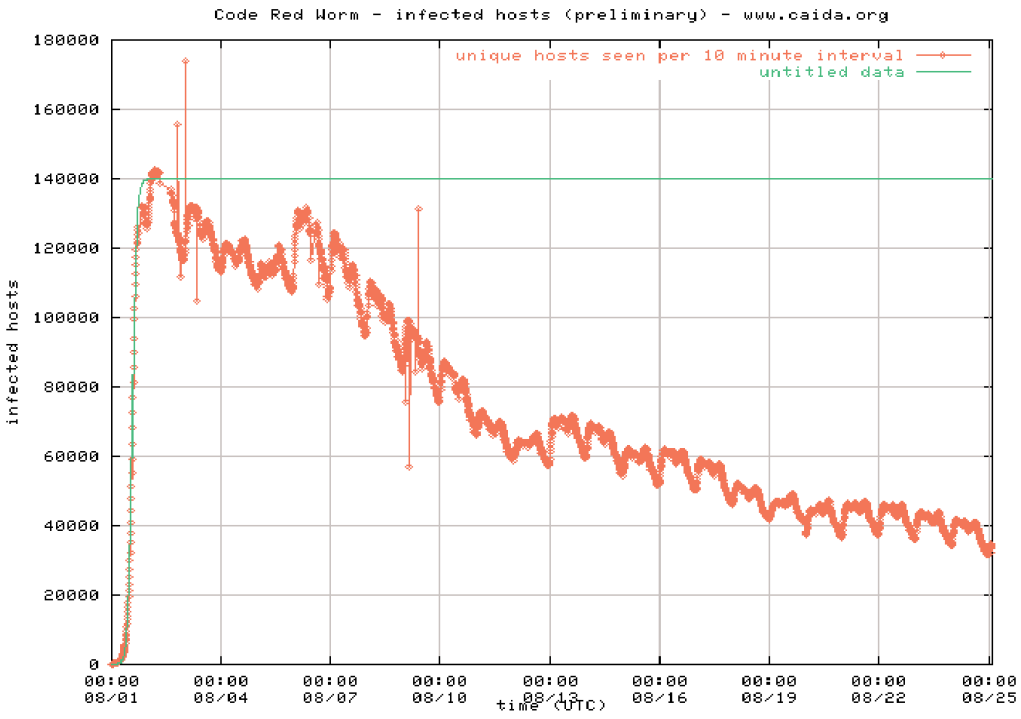
Clean-up means deleting all infected files and restoring from backups. On an infected network of gullible users this could be quite an overhead.

Other Examples

Here we will take a less detailed look at bugs in a number of other viruses, and briefly examine their impact.

Code Red Worm

The first version of the Code Red worm used a static seed when generating its target IP addresses. The result was that every computer infected with Code Red generated the same list of "random" addresses to attack. The graph below plots the unique hosts seen per 10 minute interval, the swift decrease is due to this bug.[13]



As well as crippling the spread of the worm, this bug meant that the worm ended up re-infecting the same systems time and time again. This lead to a Denial of Service due to

the amount of data being transferred between the addresses generated.

W32.Lovgate.A@mm

After setting up the variable szPassword1 as “xyz123”, the worm proceeds to check the password received as follows: (Note: *ebx* is the length of the password received)

```
.text:0040200F cmp ebx, 7
.text:00402012 jnz short loc_40202A
.text:00402014 lea eax,
[esp+1378h+szPassword1]
.text:00402018 lea ecx,
[esp+1378h+szUserRecvPass1]
.text:0040201C push eax ; char *
.text:0040201D push ecx ; char *
.text:0040201E call _strstr
```

Because the worm performs a 7 character check, this allows successful authentication with passwords of the form <any character>xyz123 or xyz123<any character>.

A few lines earlier in the code, when setting up a socket to listen on port 10168, the worm performs a double call to *socket()* ignoring the return from the first call.

W32.Logitall.A@mm

An interesting mass-mailing worm, W32.Logitall.A@mm goes to pains to print accurate debug information to the local log file in C:\MyV_DIR\MyV.log, but unfortunately the author didn’t take such care in ensuring his registry key strings were accurate. By forgetting to double-backslash his paths, the attempts to set *Run* subkeys fail.

```
3036 && 3092 && 11 && 20040526 && 15:18:08 &&
Adjusting Registry...
3036 && 3092 $$ 12 && 20040526 && 15:18:14 &&
MyV2.cpp:66 && Could not open key
SOFTWARE\Microsoft\Windows\CurrentVersion\Run
3036 && 3092 $$ 13 && 20040526 && 15:18:14 &&
MyV2.cpp:66 && Could not open key
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Run
3036 && 3092 $$ 14 && 20040526 && 15:18:14 &&
MyV2.cpp:66 && Could not open key
SOFTWARE\Microsoft\Windows NT\CurrentVersion\Run
3036 && 3092 && 15 && 20040526 && 15:18:14 &&
Good... I’m alone! Continuing...
3036 && 3092 $$ 16 && 20040526 && 15:18:14 &&
main.cpp:133 && Semaphore SemCurrDir963423093 criada
```

VBS.SST@mm

This clever piece of code is designed to recreate the malicious script file if it has been deleted. Instead it created a 0 byte file of the same name. Nice one.

```
Set opentextfile= filesystemobject.opentextfile(wscript.scriptfullname, 1)
opentextfilereadall= opentextfile.readall
opentextfile.Close
Do
If Not (filesystemobject.fileexists(wscript.scriptfullname)) Then
Set createfile= filesystemobject.createtextfile(wscript.scriptfullname, True)
createfile.writeopentextfilereadall
createfile.Close
End If
Loop
```

VBS.Pet_Tick.N

On opening a file for infection, VBS.Pet_Tick.N checks if the first line is the string “<Lover>”. This appears to be a check for an infection marker, however just 4 lines later we see it writes the string “<dilan>” to the file as if that was intended as the marker. The result being that the virus can re-infect the same file over and over.

```
Set gd=fso.OpenTextFile(cible.path,1)
If gd.readline <> "<Lover>" Then
htmorg=gd.Readall
gd.Close
Set gd=fso.OpenTextFile(cible.path,2)
gd.WriteLine "<dilan>"
gd.Write(htmorg)
gd.WriteLine document.body.createtextrange.htmltext
```

W32.Beagle.BH@mm

This beagle variant adds a value corresponding to its filename to the following registry keys, in what appears to be an attempt to ensure they are executed on startup:

```
HKEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Ru1n
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Ru1n
```

The same misspelt strings are also used in its attempt to remove the Run keys of a number of anti-virus and security related products.

W32.Mytob.MK@mm

This “Mytob” variant manages to find the Run registry subkey correctly, and sets the value:

```
“WINDOWS” = “\jif.exe”
```

However, the worm does not create a copy of itself as jif.exe in the system path so the attempt to execute the file on boot will fail. This variant also includes code to password-protect and zip the email attachments that it sends, but the author opts not to use it.

Why so many Bugs?

Bug-ridden viral code proves that you do not need to be an excellent programmer to produce some very damaging and dangerous malicious threats. Although much maligned for their lack of technical skill, at times, the above examples prove that even though the code is less than perfect it still poses serious threats to our computer networks. In this regard, there is little motivation in most cases for the virus author to perfect his code before release.

However, this doesn't mean that virus writers are necessarily bad coders. They still consistently come up with ways around the latest and greatest security procedures and products. There seems to be a premium on innovation and creativity in the virus world, where in general authors will ensure their newest pieces of code work well, and sometimes overlook the basics while caught up in the excitement of their new techniques.

The lack of resources to test releases properly has always been a huge barrier to successful virus authors, and there is no doubt that if they put their release through the full software development cycle we would see some extremely dangerous virulent code emerging regularly.

The Professional Era

Recently, with the advent of professional production teams working on malicious code for profit we are seeing increases in the effectiveness of code released, and more advanced techniques being used more often. Professional Spyware applications and Banking Trojans raise the bar in terms of code quality and testing.

More targeted attacks see threats developed for a specific environment where they have been tested, but if run on other systems may not operate at all as intended.

An Analysts Approach

When analyzing a sample it's best to assume that it would function as intended, perhaps on systems different to those the analyst is currently working on. Analysts should endeavor to provide customers with as much information as possible in the early stages of analysis and if it transpires that part of the functionality of the threat is never activated then that information can be released as it is found.

However, while it is important that analysts release as much information about a threat as possible so that customers are protected, it is also vital that they do not overstate the potential risks and cause unnecessary panic.

References

1. http://en.wikipedia.org/wiki/Mariner_1
2. http://en.wikipedia.org/wiki/Therac-25
3. http://www.securityfocus.com/news/8032
   http://www.securityfocus.com/news/8016
4. The Internet Worm Program: An Analysis, Eugene Spafford (November 28, 1988)
5. http://web.mit.edu/eichin/www/virus/main.html
6. Sobig, Sobigger, Sobiggest, by Peter Ferrie, from Virus Bulletin(October 2003).
7. http://en.wikipedia.org/wiki/Sobig
8. http://sunbeltblog.blogspot.com/2006/01/blackworm-worm-over-18-million.html
8. http://www.theregister.co.uk/2006/02/16/mac_os-x_virus/
10. http://www.sophos.com/pressoffice/news/articles/2006/02/macosxleap.html
11. http://technology.guardian.co.uk/online/security/story/0,,1712275,00.html
12. http://www.techweb.com/wire/security/180203187
13. http://www.caida.org/analysis/security/code-red/

About Symantec
Symantec is the global leader in information security, providing a broad range of software, appliances, and services designed to help individuals, small and mid-sized businesses, and large enterprises secure and manage their IT infrastructure. Symantec's Norton™ brand of products is the worldwide leader in consumer security and problem-solving solutions. Headquartered in Cupertino, California, Symantec has operations in 35 countries. More information is available at www.symantec.com.

Symantec has worldwide operations in 35 countries. For specific country offices and contact numbers, please visit our Web site. For product information in the U.S., call toll-free 1 800 745 6054.

Symantec Corporation
World Headquarters
20330 Stevens Creek Boulevard
Cupertino, CA 95014 USA
408 517 8000
800 721 3934
www.symantec.com

04/05 10406630