Security Response

32-Bit Virus Threats on 64-Bit Windows

Atli Guðmundsson
Senior Software Engineer
Symantec Security Response

Contents

Abstract

Traditionally, Intel and Microsoft have been backward compatible with older versions of their products. The IA64 architecture and the Windows XP 64-bit operating system are no exceptions to this rule, taking elaborate steps to make old applications work seamlessly under this new combination.

These new platforms, however, present an interesting risk, a risk that needs to be taken seriously. This risk is existing malware, which runs on the current 32-bit Windows platform.

This paper will explain how the Win64/IA64 platform supports Win32/IA32 applications and therefore already existing malware for that platform, but will also investigate how current malware can potentially corrupt and destroy Win64 applications.

Finally, this paper will present guidelines for AV vendors on how to recognize these types of corruptions as well as possible ways of fixing them.

Compatibility

Microsoft and Intel have put much effort into making the transition between the 32-bit environment and the 64-bit environment as smooth as possible, both in regards to the developer experience and the user experience. They have achieved this by introducing functionality that allows users (and vendors) to use existing 32-bit applications and components on Win64 and also by assisting and supporting developers in such away as to make porting as smooth as possible.

The introduced (supporting) functionality is an emulation environment called ‘Windows on Windows 64’ (WOW64). This environment acts as an intermediate layer between the 32-bit application and the 64-bit kernel, translating any API calls from the 32-bit application before they are passed on to the 64-bit kernel.

WOW64 only emulates the API calls (with some restrictions, as we will see later) and not the actual 32-bit instruction stream, acting as an intermediate medium between the 32-bit process and the 64-bit kernel. The 64-bit CPU itself does the task of supporting the actual 32-bit executable code, keeping performance penalties to a minimum. The only thing WOW64 does in that regard is switch between the two modes of operation (i.e. 32-bit or 64-bit).

When executing 32-bit applications on Win64, some notable differences become apparent, specifically in regard to how 32-bit and 64-bit applications differ in the way they utilize the operating system.

First, 32-bit applications are not allowed to load a 64-bit DLL, and vice-versa. Secondly, during execution, 32-bit applications are isolated from accessing parts of the registry as well as parts of the file system. This means that 32-bit applications will actually see the system in a different (more restricted) way than 64-bit applications.

The registry key HKEY_LOCAL_MACHINE\SOFTWARE, as viewed from a 32-bit application, is actually not the same registry key as the one viewed by 64-bit applications. Instead, WOW64 redirects 32-bit applications to use the registry key HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node, which is accessible from 64-bit applications as well (but directly as the Wow6432Node key).

The registry key HKEY_CLASSES_ROOT (Classes) actually resides under the registry key HKEY_LOCAL_MACHINE\SOFTWARE and is required for hooking extensions and COM interoperability. Ordinarily this would mean that 32-bit applications could not hook extensions (allowing the application to be launched when a user opens a document with that extension, from Explorer) or use COM interoperability. For example, if a 32-bit application added a subkey to HKEY_CLASSES_ROOT to hook a file extension, the key would be redirected to the Wow6432Node. Thus, when 64-bit Explorer is utilized to open file, the appropriate registry keys for file extension association would not be under HKEY_CLASSES_ROOT, but under the Wow6432Node instead and the hook would not work properly.

However, to facilitate this functionality as well as some interoperability for COM and other mechanisms WOW64 actually replicates the Classes subkey between the Wow6432Node hierarchy and the original registry location. This replication gives 32-bit applications the ability to hook extensions as well as using or registering COM objects just as if they were running on a 32-bit NT system.

Note that WOW64 also replicates the key HKEY_CURRENT_USER\Software\Classes in this manner.

For the run keys things are a little bit different. First, any run key that resides under the

HKEY_LOCAL_MACHINE\SOFTWARE key will not be replicated. This obviously means that 32-bit applications cannot use that subkey to execute themselves during startup. However, the run keys that reside under HKEY_CURRENT_USER\Software are not redirected by WOW64, so 32-bit applications that use those keys will not be affected.

For accessing the file system, the architecture is a bit simpler:

Any time a 32-bit application tries to access the folder %SystemDrive%\System32, the emulation layer will intercept the request and redirect it to the %SystemDrive%\SysWOW64 directory.

The files that reside in %SystemDrive%\System32 are actually the 64-bit system files and are intended for use by 64-bit applications. This file redirection was done to facilitate ease of porting for existing 32-bit code to the new platform; therefore all of the 64-bit DLLs also have the same names as the 32-bit DLLs. This means that the 32-bit DLLs have to reside somewhere else and Microsoft’s choice was %SystemDrive%\SysWOW64. The %SystemDrive%\SysWOW64 folder therefore contains all 32-bit system DLLs required for 32-bit application use. The folder essentially contains the same DLLs that a normal 32-bit NT system does (the actual code in the DLLs is different though).

For example, let's say a user wanted to use a 32-bit file viewer to view the file C:\WinXP\System32\CALC.EXE (assuming %SystemDrive% equals C:\WinXP). When the application gives the open request to the system, the system will simply redirect it to the path C:\WinXP\SysWOW64\CALC.EXE instead (it will not even work if the 32-bit application is located in the C:\WinXP\System32 folder).

This has some implications for 32-bit applications that attempt to reside in the %SystemDrive%\System32 folder and create a run key that points to themselves at that location. The run key path will be set to %SystemDrive%\System32, but the executable will actually reside in the %SystemDrive%\SysWOW64 folder due to the WOW64 file system redirection. On startup, the system will attempt to load the 32-bit application file from %SystemDrive%\System32 (since the system, which does the loading, is a 64-bit process and doesn't have any file redirection restriction), but will fail since the 32-bit application file resides in %SystemDrive%\SysWOW64. 32-bit applications can, however, use an API to recognize that they are running on Windows XP 64 and based on that information locate the correct path to themselves within the SysWOW64 folder, using another API.

This redirection should place some pressure on antivirus vendors to produce 64-bit versions of their products, especially since the 32-bit versions will be unable to access the %SystemDrive%\System32 folder, which can be very dangerous (see below).

This file redirection is not complete though. Specifically, if the System32 folder or any of its parent folders are shared or substituted, then this path redirection will not be in place (systems that have the default C$ share open will be particularly vulnerable to 32-bit viruses such as W32.Funlove).

This file redirection is only for the %SystemDrive%\System32 directory and does not extend to the Program Files folder, which commonly holds other applications.

However, 32-bit applications will usually use or give the folder 'X:\Program Files (x86)' as a default installation path, since WOW64 intercepts the %ProgramFiles% environment variable of a 32-bit process and sets it to 'X:\Program Files (x86)'. The %ProgramFiles% environment variable for 64-bit processes is not intercepted so 64-bit applications will use the 'X:\Program Files' folder as a default installation path (X in the preceding examples is an arbitrary drive letter).

Aside from the above noted differences, the 64-bit Windows operating system supports 32-bit applications just as if they were running on a 32-bit NT system (specifically this includes 32-bit console, GUI and service applications).

To make coexisting easier still for 32-bit and 64-bit applications, WOW64 supports interprocess communication between those processes using the following mechanisms:

- Handles to named objects such as mutexes, semaphores and file handles
- Handles to windows
- Remote Procedure Calls (RPC)
- COM LocalServers
- Shared memory (e.g. using file mapping)

Furthermore, the CreateProcess and ShellExecute APIs can launch 32-bit and 64-bit applications, regardless of whether the running application is a 32-bit or a 64-bit process (which turns out to be ideal for HLLP viruses, other prepending viruses and those that alter the MZ.offset_of_new_header field to point to a new PE header).

Finally, any 32-bit application, with the appropriate access rights, can view and modify any other 32-bit processes address space and although 32-bit processes can't see inside of 64-bit processes, they can still iterate and remove them from memory.

Examples

Examples of 32-bit applications that do work under Win64 are REGEDIT.EXE, CMD.EXE, CACLS.EXE (an access rights manipulator) and FTP.EXE (all taken from a 32-bit NT system), along with most FTP server applications (e.g. War-Ftp).

These show that 32-bit applications will have no problems in modifying and viewing the registry, executing other applications, modifying access rights and connecting to the Internet both as a sender as well as a receiver.

This of course implies that most malware that currently executes under 32-bit NT systems (NT4, 2000 and XP) today will execute properly under 64-bit Windows.

This includes such things as Trojan horses, backdoors, worms and, of course, file infectors.

Other things that are supported fully on Windows XP 64-bit are 32-bit Visual Basic applications and scripts, such as JavaScript (.JS), VBScript (.VBS) and Batch (.BAT/.CMD) scripts.

Changes

So far, Win64 seems to support many things, but some things have actually been removed and there are some not so well documented changes.

The most obvious thing missing is the support for legacy DOS applications, but Microsoft has finally removed support for the 16-bit DOS application as well as the 16-bit Windows application (CMD is still in there to support console applications).

If a user copies a 16-bit file over to the system and tries to execute it, Windows XP 64 will simply display a dialog box that indicates that the file is not supported for execution.

Of course, this means that developers will have to upgrade all of their DOS utilities before they start working on that platform (for example, many developers keep old DOS versions of grep and make that they are particularly fond of).

Among those applications that have disappeared are the applications DEBUG.EXE and EDLIN.EXE, which have existed in all versions of DOS and Windows up to this point.

But, for those who like doing things from the command line, the NT Symbol Debugger (NTSD.EXE), which is included in NT 2000 and XP systems (including Windows XP 64), is an excellent choice. NTSD is able to step into all 32- and 64-bit applications, both in kernel and user mode.

Now, even though support for 16-bit applications has been removed, there are still some remnants left in the system. For example, the extension .EXE can be replaced with .COM or .PIF to no ill effect for the affected application.

32-bit context menu handlers are also not supported under Win64. This is simply due to the fact that context menu handlers are loaded into the process space of EXPLORER.EXE and since these context menu handlers are 32-bit applications (or DLLs rather) then they cannot be loaded by Explorer (a 64-bit process).

Another noteworthy limitation, which Microsoft has actually worked around, is the support for ActiveX components. Since ActiveX components are 32-bit DLLs they cannot be loaded into the address space of 64-bit Internet Explorer (IE). Instead, Windows XP 64-bit comes bundled with 32-bit IE, which takes care of the 32-bit ActiveX components.

Finally, the base address for the 32-bit Kernel32.dll under Windows XP 64 is different from any of the other 32-bit platforms. Since some viruses use hard coded base addresses to find the kernel, this alone will prevent those viruses from executing on the platform.

PE+

The difference between 32-bit and 64-bit applications does not only lie in how they can utilize the system, there is also another more subtle difference.

This difference has to do with the file format that the executables reside in. This file format, known as the PE format, has been extended to encompass the enormous memory space now available to a 64-bit application. This allows a 64-bit application to specify greater size for its stack and heap as well as giving a developer a greater range of choices as to where to load their applications and DLLs in the linear memory space.

Due to the extension of the file format, 32-bit Windows viruses have the potential to cause more indirect damage than usual on the 64-bit Windows platform.

This indirect damage will occur regardless of whether the virus is running locally or infecting remotely via an open share. The reason for this is the slightly different PE specification for 64-bit applications (hereafter referred to as PE+).

This difference is mostly structural in nature and, aside from the Base of data field (which has been removed from PE+), does not change the functionality of any field or structure.

These differences are specific to the PE optional header, the import lookup/address table, the export address table and the TLS directory.

Aside from these structural differences (where the size of some fields has been increased from four to eight bytes), there are also additional informational differences in the content of three key fields. These key fields are the Machine signature field (at offset 4h), the size of optional header field (at offset 14h) and the optional header signature field (at offset 18h).

Table 1

PE Header overview

Offset [PE+]	Size [PE+]	Description
0h	4h	PE Signature (image only)
4h	14h	COFF file header
18h	60h [70h]	Optional header (usually image only)
78h	[88h] Variable (80h)	Optional header pointer/size pairs to various data tables (usually image only) (Size is indicated by the optional header, but 80h is standard for most compilers)
F8h	[108h]	Variable Section table (size is indicated by the COFF header)

The machine signature field indicates the intended CPU platform – 014Ch is for the IA32 architecture and 0200h for the IA64. Other values are of course possible, but are not important for this paper.

The most common value for the size of optional header field changes from E0h to F0h (though it can be as small 70h and 88h for 32-bit DLLs and executables, respectively), due to the 4 byte increase in size of the fields Image Base, Size of Stack reserve, Size of Stack commit, Size of Heap reserve, Size of Heap commit as well as the removal of the Base of data field. Note that, since the size of optional header field also includes the optional header pointer/size pair table (which includes pointers to imports, exports, resources, etc.), it does not have a fixed value. However, most compilers set it to E0h or F0h (PE+).

Finally, the optional header signature field indicates the format of the optional header – 010Bh signifies the PE structure and 020Bh signifies the PE+ structure. (Older versions of 32-bit Windows only check the Machine signature field, but Windows XP 64 checks the optional header signature field as well.) Of the three fields, the optional header signature field is the only one that indicates to applications that the file format is different.

For a closer look at the structure of the PE header and a comparison between the old and the new offsets, please refer to Appendix A.

Assumptions

One would assume that 32-bit viruses (or their creators) were conscious enough not to infect 64-bit applications, because they are incompatible. After all, 64-bit applications contain 64-bit Intel-specific machine code as opposed to the 32-bit Intel-specific machine code for the 32-bit platform. In addition, the file format is different.

As mentioned previously, there are two fields (three if you count the size of optional header field) that give a clear indication that the content of the image file is different; one gives the type of CPU instructions used and the other the actual PE structure type.

But, upon closer inspection, nothing has seemingly changed in the PE+ specification, all structures accessible through the PE header (with the exception of the TLS and import tables) are the same, performing exactly the same function. The only problem is locating them correctly.

For the section table the correct way is to use the size of optional header field.

For locating the various tables correctly, the viewer has to examine the optional header signature field for indication of which format this particular PE header follows, to determine where the tables actually start (and to be able to locate the number of optional pointer/size pairs field). However, things are not so simple.

First, not all viruses infect in the same manner. Some are non-intrusive to the application being infected, not actually inserting themselves into the hosts file structure. Companion viruses, most high-level language (HLL) viruses, as well as some worms fall into this category, usually restoring the host completely before passing control to it.

Secondly, viruses are not perfect programs. They generally have a high percentage of bugs and have, usually, a low tolerance for any differences in functionality or implementation of the platforms they generally execute on.

A good example of a difference that some viruses have a low tolerance for is the base address of the Kernel32.dll, which changes with each new version of Windows (as mentioned earlier viruses often try to locate that DLL using hard coded reference pointers, which in this case will prevent them from executing on the platform).

Another example would be the NT platform itself. Virus writers need considerable time to understand the restrictions NT enforces on running applications. Thus, many viruses just crash when executed under Windows NT.

Today, most viruses use structured exception handling (SEH) to guard against these restrictions, passing control silently to the host when encountered. Viruses under 64-bit Windows will encounter similar problems.

The following are the validation steps that a typical 32-bit Windows virus goes through before it infects another application:

1. Make sure the application starts with an MZ stub (a requirement for PE applications). This is done by making sure that the first two bytes of the file are ‘M’, ‘Z’.
2. Next, locate the PE header by using an offset field in the MZ stub and make sure it actually points to a PE header (the header starts with the bytes ‘P’, ‘E’, ‘\0’, ‘\0’).
3. Check for any infection marker(s) (most viruses set some supposedly unused field or file location to some magic value; some even check if the file size matches some criteria [e.g. if it divides evenly up into some number]).

Step two is mandatory for a 32-bit Windows virus that inserts itself into the host, since not all MZ executables are PE files, they could be a plain vanilla DOS executable, a 16-bit Windows application (NE) or a linear executable (LE/LX). Of course, there are a handful of viruses that ignore this step and manage to corrupt these other executable types.

Other checks or self imposed limitations used by virus writers have been the following:

1. Make sure it is a valid executable (i.e. test whether the image characteristics has the valid bit set).
2. Verify that the potential host is not a Dynamic Link Library (DLL).
3. Verify that the base address is a particular value (e.g. 00400000h).
4. Verify that the required machine type/platform is 014Ch, which indicates Intel 32-bit.

Then, after deciding that the application is suitable for infection, the virus typically goes through the following (simplified) steps:

1. Locate entry point to infect (for entry point obscuring (EPO) viruses this is commonly an import call or a specific piece of code within the application image).
2. Write itself to the executable file.

To write itself to the file, a virus needs to determine what the structure of the file is, where the code is, and where it can hook and insert itself into the host. This process in almost all instances (except for pure header infectors) requires the use of the section table. This is where many viruses make a fatal mistake.

Sections

To better understand the issue, we should take a brief look at what information the section table provides about the executable image to the operating system, as well as to other applications that wish to view the content of the image file. We should also try to understand what steps should be followed to locate and acquire information from the section table.

The section table quite simply contains information about how the physical image file and the subsequently loaded image are related. In other words, it contains information about how particular regions of the application memory image are filled with data from the image file.

The steps to locate this table are quite simple. One takes the PE header offset (as a VA, RVA or physical offset), the size of the COFF header (which is constant), the size of the PE signature field (also constant) and the size of the optional header and adds these values together. This will result in the offset of the section table. This is the only correct way of locating this table, since the optional header size is not necessarily a constant value between applications (nor is it so according to the specification).

The section table consists of a number of entries (as indicated by the PE header) that describe one section in the image file.

Table 2

Section table entry

Offset	Size	Description
0h	8h	Section name (ignored by the operating system)
8h	4h	Virtual size of the section
Ch	4h	Relative Virtual Address (to the image base) of the section
10h	4h	Physical Size of the section in the image file
14h	4h	Physical Offset of the section in the image file
18h	4h	Pointer to relocations (usually 0 for executable images)
1Ch	4h	Pointer to line numbers (usually 0 for executable images)
20h	2h	Number of relocation entries (usually 0 for executable images)
22h	2h	Number of line number entries (usually 0 for executable images)
24h	4h	Characteristics of the section (readable, writeable, executable, etc.)

Problem

Let’s go back and take a brief look again at the steps the common virus goes through when deciding whether to infect an executable or not. There is something wrong with the list.

As you can see, the typical virus does usually not perform a check against the indicated platform. In fact many new viruses still don’t include this test and a recent example of such a virus is the W32.ElKern family of viruses.

This means that a virus that goes through the first five steps will infect a 64-bit application just as if it were a 32-bit application. Yet again a recent example of such a virus is W32.ElKern.

Another fatal mistake for the 32-bit viruses is the new size of the optional header in 64-bit applications.

As we can see an application should always grab for the size of the optional header from the COFF header. However, a few viruses just assume that the size of the optional header is E0h bytes in size.

An example of a virus that makes this kind of an assumption is W95.MTX (W32.ElKern properly uses the value from the COFF header).

This mistake leads the virus to use the section table as if it started from an offset earlier than it actually does (10h (16) bytes earlier).

Again, this causes the virus to interpret the structure of the potential host wrongly such that when it tries to infect the host, it will become corrupted if the virus code doesn’t recognize that something is going wrong during infection. In either case one ends up with a corrupted PE+ executable.

In the first case, the structure of the executable is preserved, but now contains 32-bit Intel-specific code mixed in with the 64-bit code and in the second case the result is totally unpredictable. This will usually require a clean backup to restore the file back to a functional state.

Table 3 shows what data a virus, which assumes the PE header is E0h bytes long, will actually be fetching when looking at a file with three sections.

Hope

The outlook looks pretty grim for a Win64 platform that becomes infected with a 32-bit virus. But not all hope is lost yet.

First, files in the 64-bit system files folder should be intact, assuming that the virus didn't infect over an open share and into a folder that is in the path of the System32 folder (e.g. the C$ share), even though all other PE+ applications could potentially be infected or corrupted.

Secondly, most 32-bit viruses that do manage to infect PE+ 'correctly' by calculating the location of the section table correctly will lead to a specific type of corruption where an IA64 image file will contain IA32 code pieces from a virus.

Finally, those viruses that don't calculate the section table offset correctly, but still want to infect a PE+ executable are few and far between.

This type of corruption is very easy to handle.

When infecting the PE+ structure, the modifications done by the 32-bit virus will be just as if the image was a PE file, except in that in this case the host code is IA64-specific.

This will lead to such cases as a 64-bit executable that has a 32-bit virus inserted into the last section of the file or spread over cavities in the file, with the original entry point pointing to the start of the virus.

Table 3

Actual vs. assumed fields for 3 sections

Actual Field Values	Virus [Incorrect] Field Assumptions
.NET RVA/size pair	#1 Section name
Reserved Table Fh RVA	#1 Virtual size
Reserved Table Fh size	#1 Relative Virtual Address
Actual #1 Section name [0…3]	#1 Physical Size
Actual #1 Section name [4…7]	#1 Physical Offset
Actual #1 Virtual size	#1 Pointer to relocations
Actual #1 Relative Virtual Address	#1 Pointer to line numbers
Actual #1 Physical Size [0…1]	#1 Number of relocation entries
Actual #1 Physical Size [2…3]	#1 Number of line number entries
Actual #1 Physical Offset	#1 Characteristics
Actual #1 Pointer to relocations	#2 Section name
Actual #1 Pointer to line numbers	
Actual #1 Number of relocation entries	#2 Virtual size
Actual #1 number of line number entries	
Actual #1 Characteristics	#2 Relative Virtual Address
Actual #2 Section name [0…3]	#2 Physical Size
Actual #2 Section name [4…7]	#2 Physical Offset
Actual #2 Virtual size	#2 Pointer to relocations
Actual #2 Relative Virtual Address	#2 Pointer to line numbers
Actual #2 Physical Size [0…1]	#2 Number of relocation entries
Actual #2 Physical Size [2…3]	#2 Number of line number entries
Actual #2 Physical Offset	#2 Characteristics
Actual #2 Pointer to relocations	#3 Section name
Actual #2 Pointer to line numbers	
Actual #2 Number of relocation entries	#3 Virtual size
Actual #2 number of line number entries	
Actual #2 Characteristics	#3 Relative Virtual Address
Actual #3 Section name [0…3]	#3 Physical Size
Actual #3 Section name [4…7]	#3 Physical Offset
Actual #3 Virtual size	#3 Pointer to relocations
Actual #3 Relative Virtual Address	#3 Pointer to line numbers
Actual #3 Physical Size [0…1]	#3 Number of relocation entries
Actual #3 Physical Size [2…3]	#3 Number of line number entries
Actual #3 Physical Offset	#3 Characteristics

For those familiar with how 32-bit viruses infect a PE executable you will find it easy to create other scenarios.

Now, the reason this will work lies in the fact that most viruses do not affect or use any of the fields in the PE header that reside after offset 5Eh (DLL characteristics). Those that do are commonly EPO viruses that use the location of the import table as a helper in choosing which potential calls in the image are actually references to system APIs, which of course results in different problems.

Most current virus scanners will only need to be modified slightly to be able to deal with this class of viruses (basically assume the host is a normal PE executable), since the virus scanner can use already existing code for everything else. In fact, tests indicate that some antivirus products already show this behavior, but whether this is intentional or simply because they are not checking the file format correctly shall be left unsaid.

Risks

This paper would not be complete without some kind of a risk assessment in regards to a handful of the current most prevalent 32-bit Windows viruses and worms that actually infect and corrupt 64-bit Windows executables.

The following pages list the risks with some of these binary worms and viruses, using the prevalence list as reported by Virus Bulletin magazine in their issues from February 2002 through June of 2002 (virus prevalence tables for December 2001 through April 2002).

W32.Nimda.A@mm

IA64 file risk
High

Executes on IA64
- Yes, is able to infect files while running on the platform.
- Is able to install itself as a service just as under 32-bit NT.

Detection/Repair
Same as on IA32 (unmodified).

File checks
- Extension must be .exe.
- File name must not match winzip32.exe.
- Byte at file offset D6 must not match the value 8F.

Infection notes
Original host is stored as a resource in the virus.

W32.Klez@mm

IA64 file risk
High

Executes on IA64
Yes, will infect files on the system without any problems.

Detection/Repair
Same as on IA32 (unmodified).

File checks
- File does not contain a unique marker string.
- The System File Checker (SFC) is not protecting the file.
- MZ signature is equal to ‘M’, ‘Z’.
- First four bytes of PE signature are equal to ‘P’, ‘E’, ‘\0’, ‘\0’.

Infection notes
Creates a hidden [compressed/encrypted] backup of the original host file and then writes over the host with itself.

Other notes
Carries W32.ElKern.4926 within itself and inserts it onto the system when executed.

W32.ElKern.4926

IA64 file risk

- Almost nonexistent (due to bugs in the virus).
- During infection W32.ElKern.4926 tries to validate that no relocations will be applied to the entry point when the image is loaded, however due to the fact that the virus uses a hardcoded offset to locate the relocation table in the PE header, it accidentally fetches the offset for the exception table instead (this table is present in most IA64 executables). Since the virus is now parsing invalid data (as far as relocations are concerned) it more often than not causes an exception fault, which prevents it from infecting that file.

Executes on IA64

Yes.

Detection/Repair

Must assume that the file is a 32-bit PE file.

File checks

- File name does not start with AVP, NAV, _AVP, ALER, AMON, ANTI, NOD3, NPSS, NRES, NSCH, N32S, AVWI, SCAN, F-ST, F-PR.
- Extension of file is .exe or .scr.
- Path of file does not contain ‘tem32\dllcac’ (as in System32\dllcache).
- MZ signature is equal to ‘M’, ‘Z’.
- First two bytes of PE signature are equal to ‘P’, ‘E’.
- Image is not a DLL.
- Subsystem is GUI or CUI (PE.SubSystem equal to 2 or 3).
- Original file size is greater or equal to 8192.
- File does not contain the string ‘irus’ anywhere in its body (after the PE header).
- The System File Checker (SFC) is not protecting the file.
- Section two does not contain the string ‘WinZip’ at offset 10h, as in SFX WinZip archives (the virus assumes there are at least two sections).
- Last section does not start with the string ‘Rar!’, as in SFX Rar archives.
- Entry point is not preceded with the string ‘WQ’ (infection marker).

Infection notes

- Always increases the size of the file by 9,030 bytes.
- Calculates the offset of the section table correctly.
- Hooks the entry point, using one of two methods:
- Changes the first five bytes at the entry point into a jump instruction, which transfers control to the virus body.
- If there are relocations to the bytes at the entry point then the virus changes the entry point to point to itself.
- Spreads the virus body over cavities in the file, increasing the size of the last section if needed.

W32.Magistr.39921@mm

IA64 file risk

Almost nonexistent (due to bugs in the virus) and then only across open shares.

Executes on IA64

No, will crash due to bugs in the virus (assumes the base address of Kernel32.dll to be something other than what it actually is).

Detection/Repair

- Must assume that the file is a 32-bit PE file.
- Must assume that the section table is at the same location as in standard PE header (since the virus doesn’t correctly calculate the offset of the section table).

File checks

- Extension of file is .exe or .scr.
- No check for MZ stub, instead the offset of the PE header is checked to be within the file.
- First four bytes of PE signature are equal to 'P', 'E', '\0', '\0'.
- Image is valid (is executable).
- Image is not a DLL.
- Subsystem is not Native nor Console (will try to infect any other, including GUI and EFI).

Infection notes

- Does not calculate the offset of the section table correctly.
- Tries to overwrite the data at the entry point with a polymorphic transfer routine, which transfers control to the virus decryptor.
- If it successfully writes to a PE+ file (very rare indeed) then it is almost sure to corrupt the file beyond any repair.

W32.Mylife@mm

IA64 file risk
Minimal (only due to payload).

Executes on IA64
Yes (is a VB application).

Detection/Repair
Same as on IA32 (unmodified).

File checks
N/A

Infection notes
N/A

W95.CIH

IA64 file risk
Low (only across mapped drives).

Executes on IA64
Yes, but will only pass control directly to the host (uses specific Win9x methods that prevent it from ever doing anything on the system).

Detection/Repair
Must assume that the file is a 32-bit PE file.

File checks

- Extension of file is .exe.
- Note CIH does not carry out any checks for the MZ signature.
- Byte in front of PE signature must be '\0' (some variants of this family test two bytes in front).
- First three bytes of PE signature must be 'P', 'E', '\0' (some variants only test the first two bytes).
- Cavity after section table in header, upto the end of the header section, must be larger than at least 184 bytes (later variants have a requirement of upto 238 bytes).
- Later variants also verify that section two does not contain the string 'nZip' at offset
- 12h, as in SFX WinZip archives (the virus assumes there are at least two sections).

Infection notes

- Calculates the offset of the section table correctly.

- Inserts itself into the header of the file, using cavities in the remainder of the file for parts that do not fit into the header.
- Changes the entry point to itself.

W32.Funlove.4099

IA64 file risk
Low (only across shares).

Executes on IA64
Yes, but will only pass control directly to the host (to find the kernel base address it uses hardcoded values, which do not include the base address of the 32-bit kernel under Windows XP 64).

Detection/Repair
Must assume that the file is a 32-bit PE file.

File checks

- File name does not start with ALER, AMON, .AVP, AVP3, AVPM, F-PR, NAVW, SCAN, SMSS, DDHE, DPLA or MPLA.
- Extension of file is .exe, .scr or .ocx.
- Size of file is at least 8,192 bytes.
- Size of file module 256 is not equal to 3 (infection marker).
- MZ signature is equal to ‘M’, ‘Z’.
- MZ relocation offset is equal to 40h.
- The offset of the PE header is somewhere in the first 7168 bytes.
- First two bytes of PE signature are equal to ‘P’, ‘E’.
- Subsystem is GUI (PE.SubSystem equals 2).
- Last section uninitialized flag cleared.
- Last section starts within the first 87.5% of the file or the section covers at least all the bytes up to the end of the first 87.5% bytes.

Infection notes
- Calculates the offset of the section table correctly.
- Hooks the entry point by changing the first eight bytes to the instructions nop, nop, fs:jmp $xxxxxxxx.
- Attaches itself to the last section.

Solutions

We can now see that 32-bit malware is a potential threat to Windows XP 64 and specifically that 32-bit viruses are likely to infect and execute on that platform with minor difficulties, potentially corrupting 64-bit executables in the process.

To tackle this problem the antivirus vendor needs to do a few things:

- The vendor needs to create a 64-bit scanner, since otherwise a part of the file system becomes inaccessible to their product.
- The vendor needs to take added care when dealing with side effects of viruses, since some modifications to the registry are not replicated from the 32-bit registry tree to the 64-bit one (and vice-versa).
- The vendor needs to change their scanner such that it scans 64-bit applications as if they in fact contained 32-bit executable code (emulating through the main entry point, looking for possible EPO calls, etc.), making sure that the scanner uses the size of optional header field when locating the section table.
- Obviously, the vendor also needs to support the 64-bit code base when those threats start to emerge.

For the Administrator the biggest threat comes from open shares and those threats that auto-execute when viewed in email (such as Klez and Nimda), therefore should:

- Make sure that no 64-bit server actually has a vulnerable share, protecting sub folders with access restrictions if needed.

- Never use a server to view any mail.

It is obvious that even though the PE specification was designed for 32-bit applications it has now proved itself as a truly portable executable format, allowing 32-bit viruses to infect and corrupt 64-bit files with no problems at all.

Finally, to enhance the security of the 64-bit platform in the future I would like to suggest to Microsoft that changing the first two bytes of the PE+ header, from ‘P’, ‘E’ to something else, is a strong move that will at the very least prevent most 32-bit file infectors in the future from actively corrupting 64-bit Windows applications.

Appendix A

PE header layout

[Reference: MSDN Library – October 2001]

Offset [PE+]	Size [PE+]	Description
0h	4h	PE signature (‘P’, ‘E’, ‘\0’, ‘\0’)
4h	2h	Machine signature (IA32: 0x014c, IA64: 0x0200)
6h	2h	Number of sections
8h	4h	Time/Date stamp
Ch	4h	Pointer to COFF Symbol Table
10h	4h	Number of Symbols
14h	2h	Size of optional header (System file values PE: 0xE0,
16h	2h	PE+: 0xF0) (this includes the size of the pointer/size pairtable)
18h	2h	Characteristics for the image
1Ah	1h	Optional header signature (PE: 0x010B, PE+: 0x020B)
1Bh	1h	Major linker version
1Ch	4h	Minor linker version
20h	4h	Size of code
24h	4h	Size of initialized data
28h	4h	Size of uninitialized data
2Ch	4h	Address of entry point (RVA)
30h	4h [0h]	Base of code (RVA)
34h [30h]	4h [8h]	Base of data (RVA) (this field is absent in the PE+ structure)
38h	4h	Image base (VA) (in PE+ this field overlaps the PE base of data field)
3Ch	4h	Section alignment
40h	2h	File alignment
42h	2h	Major operating system version required
44h	2h	Minor operating system version required
46h	2h	Major image version number
48h	2h	Minor image version number
4Ah	2h	Major subsystem version number
4Ch	4h	Minor subsystem version number
50h	4h	Reserved (0)
54h	4h	Image size
58h	4h	Header size
5Ch	2h	Image checksum
5Eh	2h	Subsystem required to run this image
60h	4h [8h]	DLL characteristics
64h [68h]	4h [8h]	Size of stack reserve
68h [70h]	4h [8h]	Size of stack commit
6Ch [78h]	4h [8h]	Size of heap reserve
70h [80h]	4h	Size of heap commit
74h [84h]	4h	Loader flags
78h [88h]	8h	Number of optional pointer/size pairs (set to 10h by most compilers, unused entries in the table are set to zero)

Appendix A

PE header layout

[Reference: MSDN Library – October 2001]

Offset [PE+]	Size [PE+]	Description
80h [90h]	8h	Export table (RVA, size)
88h [98h]	8h	Import table (RVA, size)
90h [A0h]	8h	Resource table (RVA, size)
98h [A8h]	8h	Exception table (RVA, size)
A0h [B0h]	8h	Security table (RVA, size)
A8h [B8h]	8h	Relocation table (RVA, size)
B0h [C0h]	8h	Debug table (RVA, size)
B8h [C8h]	8h	Machine specific table (RVA, size)
C0h [D0h]	8h	Thread local storage table (RVA, size)
C8h [D8h]	8h	Load config table (RVA, size)
D0h [E0h]	8h	Bound import table (RVA, size)
D8h [E8h]	8h	Import address table (RVA, size)
E0h [F0h]	8h	Delay import descriptor table (RVA, size)
E8h [F8h]	8h	COM+ runtime header (RVA, size)
F0h [100h]	8h	.NET (RVA, size)
F8h [108h]	Variable	Reserved (0) Section table (size is indicated by the COFF header)

This paper would, of course, not have been possible without a test system. For future reference I have included Appendix B, which includes some data about it, as given by the System Information utility on the system.

Appendix **B**

Windows XP 64 Test System

Item	Value
OS Name	Microsoft Windows XP Professional
Version	5.1.2600 Build 2600
OS Manufacturer	Microsoft Corporation
System Name	ITANIUM
System Manufacturer	INTEL
System Model	SR460AC4
System Type	Itanium (TM) -based System
Processor	ia64 Family 7 Model 1 Stepping 5 GenuineIntel ~733 MHz
BIOS Version/Date	Intel Corp. S460AC4A.86B.0083.B.0106070029, 6/7/2001
SMBIOS	Version 2.3
Windows Directory	%Windir%
System Directory	%Windir%\System32
Boot Device	\Device\HarddiskVolume1
Locale	United States
Hardware Abstraction Layer	Version = “5.1.2600.0 (xpclient.010817-1148)”
User Name	ITANIUM\Administrator
Time Zone	Pacific Daylight Time
Total Physical Memory	4,096.00 MB
Available Physical Memory	3.47 GB
Total Virtual Memory	9.73 GB
Available Virtual Memory	8.91 GB
Page File Space	5.73 GB
Page File	C:\pagefile.sys

About Symantec
Symantec is a global leader in providing security, storage and systems management solutions to help businesses and consumers secure and manage their information. Headquartered in Cupertino, Calif., Symantec has operations in more than 40 countries. More information is available at www.symantec.com.

About the author
Atli Guðmundsson worked as a Senior Software Engineer for Symantec between 2000 and 2004.

For specific country offices and contact numbers, please visit our Web site. For product information in the U.S., call toll-free 1 (800) 745 6054.

Symantec Corporation
World Headquarters
20330 Stevens Creek Blvd.
Cupertino, CA 95014 USA
+1 (408) 517 8000
1 (800) 721 3934
www.symantec.com