SANDBOXII: INTERNET

Kurt Natvig
Norman ASA, P.Box 43, N-1324 Lysaker, Norway
Tel +47 67 10 97 00 • Fax +47 67 58 99 40 • Email kan@norman.no

ABSTRACT

Last year I presented how a simulated computer, which is integrated inside the scanner engine, can detect viruses based on actual performance. I demonstrated regular file replication for regular Win32 PE infectors. However, regular file replicating viruses do not pose the biggest threat – worms and viruses spreading through the Internet do. I will demonstrate how detection of these critters can be applied to the simulated computer, how these simulated computers can 'network' inside a single scanner engine, opening shares and communicate with a simulated SMTP server, how we deal with run-time libraries, e.g. Visual Basic DLLs, what is possible to simulate and what is not.

INTRODUCTION

Worms spreading using the Internet pose the largest threat. Detecting the unknown is becoming more and more important, since you only have minutes (or even seconds) before numerous users and networks will be affected. Pushing out new signature files only *after* the virus is known isn't good enough, and hence AV researchers must implement stronger heuristics to detect things they haven't seen yet.

This has been said repeatedly. It is one thing to detect a class of viruses generically, increasing the possibility that the 'next' version of that virus will be caught. However, it is another to detect something completely unknown, written in an unknown fashion – using various possibilities of languages … How do you guard against these?

One possibility is using a simulated computer. The computer itself hasn't changed much since last year (see *Proc. Int. Virus Bull. Conf. 2001*, pp475–488 for more details). It has been expanded with networking capabilities – having somewhat limited Internet and LAN support. At first I thought you needed a lot of simulated computers to form a LAN, but to simulate it you only need one.

SIMULATING PROCESS MEMORY, MULTI-THREADING AND PRIVILEGES

To make most Win32 viruses work in a simulated computer, thread support per process must be supported by the simulated operating system. A slot of CPU cycles must be given to each thread running. The simulated OS faces the same 'problems' as the real one does with regard to overhead in management of all these threads. It's extremely easy to make threads, and viruses make remote threads in other processes. If this isn't supported, many viruses can't be detected.

Win32 applications are loaded at the preferred base address, if available. If, for one reason or another, this address cannot be used, the .reloc data (if available in the PE header) is used to relocate the application. The memory space surrounding the application is different depending on the operating system. Some libraries (e.g. Kernel32.dll) are loaded at different locations in different *Windows NT* builds and *Win 9x*. Using the PE header's ability to import gives the correct API – no matter what *Windows* operating system the application is running on. Using the API's GetModuleHandle and GetProcAddress, viruses, worms and legal applications can get the location of other APIs residing in other libraries. Using LoadLibraryA, you ask KERNEL32.DLL to load the specified DLL so that you can use it at your convenience.

There are several ways viruses can go resident in the *Windows* operating systems. The first *Windows 9x* viruses allocated memory pages and hooked the file system, using IFSMgr APIs. The new major viruses have started external threads of already running applications – giving them 'new functionality'. Many worms reside as a running process, while others might attach themselves to running processes, as explorer. To make these viruses also work on the simulated computer, our *Windows* clone operating system(s) must provide the same possibilities and APIs.

Enumerating the processes on a system can be done easily using some APIs residing on most *Windows* operating systems. *Windows NT 4* does not support these APIs, and applications using them must supply the DLL PSAPI.DLL. Our *Windows* clone has support for these in the standard

KERNEL32.DLL. First, you take a snapshot of the current state, using the API CreateToolhelp32Snapshot. This provides a handle to be used, for instance, when walking the process list, using the APIs Process32First and Process32Next. These APIs must also be simulated to provide viruses the chance of looking at the already running processes – and act thereupon (e.g. W32/Elkern.C will try to enumerate all processes to see whether it can infected them). Similar APIs exist for threads (Thread32First/Next), but these only provide the thread ID. To actually get the entry point of the code running in each thread, the kernel TEB (Thread Environment Block) must be located. The simulated computer must also cover these in its operating system. Since the application running doesn't have direct access to the memory space of other applications, APIs are provided to do this (ReadProcessMemory/WriteProcessMemory). An application trying to do so must have special debug rights.

How do you determine what rights you have? Several APIs must be provided to make viruses discover what they are in 'power' of doing. The API GetCurrentProcess will provide the handle of the running process. Using the API OpenProcessToken opens the access token associated with the process. This token can be used to retrieve specific information about the access token – user, groups, privileges and owners can be examined. Security identifiers (SID) are used to provide additional security, and must also be present in the *Windows* clone written by us.

Viruses also try to climb higher to gain a higher level of privileges than in the context in which they are already running. The same APIs and structures must give the virus the same chance inside the simulated operating system.

WHAT IS A LAN AND HOW DO VIRUSES USE THEM?

Luckily, viruses can't physically look at the computer and say – 'Hey, it's a fake'. They can't follow the physical cable and verify to whom they're actually speaking. They are relying on APIs and looking intro structures to 'explore' network capabilities. Extending the simulating computer towards LAN and Internet simulation is a matter of simulating the right APIs and filling in the correct structures.

The various 'networked' simulated computers can communicate within the LAN using given APIs. It is not connected to the real Internet – the possibility of this will be discussed later in this paper in the section '*Connect the simulated computer to the Internet*'.

A network is, as already mentioned, a series of APIs behaving as if there exists a LAN. It will provide functions to enumerate the network and use the resources. A physical network doesn't really exist (including cables and hubs to other physical computers), but that is hard for the virus to know. A network can actually be as simple as one simulated computer, having access to 'shares' on several other simulated hard disks, floating around out there in the middle of nowhere. So, for now, the local area network (LAN) will consist only of a computer, with write-access to shares and the possibility of talking to a simulated SMTP server providing the possibility of email. The *Windows* clone also contains an 'installed' Internet account (Registry settings), and dummy email addresses in the Windows Address Book (WAB) and other HTML files lying around.

How do you know what IP address corresponds to a SMTP server? Some viruses talk to 'predefined' servers or try to make up a valid name for a SMTP server. Connecting to SMTP servers is usually done on port 25, so an easy approach is to make any socket connecting through port 25 – regardless of the IP address – talk to our simulated one.

A library called MPR.DLL supplies the APIs used to explore the LAN. The WNetOpenEnum function starts an enumeration of network resources or existing connections, while the WNetEnumResource function continues the network resource enumeration. This will provide all the information necessary for a virus to enumerate the network.

To a computer, a share is just a storage device, located remotely on another system. To the user, it's just a drive, and with the correct rights, the user(s) can access them or not. Network traffic on a network volume will end up in this device driver (*Windows 9x*) rather than the VFAT driver that is installed to provide file system support on our simulated C drive. The Vredir driver must provide file system support for networked drives. If the remote drive connected to your local system is Q:, all file access going towards Q:\ will be directed by the IFSMgr towards the Vredir driver since it's a remote system. Whether this one again connects to the VFAT driver concerning another 'hidden' drive is hard for the virus to know. So, remote drives can be very local, but only accessible on the shared drive using either a mapped drive or a UNC path.

The ring3 interface against the local area network is the most important. Since most Win32 viruses reside in ring3 (application layer), these APIs must behave as the real world against the caller. What happens underneath the exported APIs is hard for the viruses to take advantage of or monitor – without a filter driver (*NT*/*2000*/*XP*) or a file system driver (*9x*).

HOW DO VIRUSES SEND EMAILS?

Viruses can send emails in a number of ways. Some contain more code than others. Many prepare the mail as a big chunk of data, and send it off to a SMTP server. Using high-level languages, like Visual Basic Scripts (VBS), the virus author doesn't need to know the details of how it works – general APIs are provided. Building the entire architecture for simulation purposes, you do need to know the details of how it works.

Basically, they open a socket against a SMTP server and send the contents they want transmitted. They must insert the commands and headers, and do some decoding if it is going to attach something binary from the already infected system. Some viruses talk to predefined SMTP servers; others carry addresses of SMTP servers they successfully communicated with earlier, and other again use the configuration in the 'Default mail account' settings in the Registry.

CONNECT THE SIMULATED COMPUTER TO THE INTERNET?

The simulated computer isn't connected to the 'real world', and cannot see whether files or IP addresses are valid or available. An option is to make callbacks available to the engine to verify addresses or download files. This must be an option, or many home users will suffer from bandwidth problems on their analogue lines, or even large corporations with thousands employees scanning their computers – image on access systems. It's more realistic a feature to put on the mail gateway, and scan mail attachments. The contents downloaded from an address could be copied into the simulated computer, and cannot touch the 'real' computer.

Another issue is testers; if they perform the test against live virus samples on a network not connected to the Internet, detection of mass mailing worms and viruses would fail. If this option is enabled, detection could be applied. Should there be an option to alert that an executable is trying to connect to a SMTP server out there? Would that be enough for detection?

A more serious problem is viruses trying to get web pages or sending HTTP GET commands towards other sites (like Nimda/CodeRed probes). These can cause the remote system to perform a buffer overrun, and would actually force them into running binary code coming from the simulated computer. It doesn't affect YOUR computer, but it could affect other computers. The callback filter must only let through validated requests, like fetching files. Security comes first. The simulated computer would never be allowed to actually upload a file anywhere on the Internet. This could be a trick viruses pull to detect simulation, but it has solutions; it can be 'stored' locally on a given IP address, and be 'available' if, later, it wants to verify its existence. On the other hand, viruses sending probes to cause buffer overruns can be detected – from the simulated computer trying to send them.

If the system isn't connected to the real Internet, a miniature can be made. Lead all IP traffic through a fake Domain Name Service (DNS) lookup server that will translate the names it knows. A simple table can be put in the signature data file, which will be updated dynamically as we proceed:

DNS	**IP address**	**Resources available**
www.norman.no	193.71.68.4	Index.htm=goat1.htm, virus_infow32_klez_g_mm.shtml=klez.htm
www.whitehouse.gov	148.122.167.38	Index.htm=goat1.htm
SMTP.playground.no	192.168.168.128	
Banners.interfree.it	213.158.72.68	List.txt=sheer.dat
Pop.eunet.no	193.75.75.100	

Viruses and legal applications can then 'talk' to the listed sites using either IP address or DNS. We can also make fake resources available at each site, so viruses can either download files or try to 'infect' them. As an example, the W32/Sheer.A virus will 'work' without being connected to the Internet, since the file it requests is available on the local Internet – of course it won't in a released version, this is just for demonstration purposes.

Viruses sending scripts of probes to other computers on the Internet cause a problem, since we only want to allow traffic *into* the simulated computers, not out from them. Image this scenario on a mail server: an attachment is coming in. This attachment is put into the simulated computers and the LAN, and starts to explore the possibilities. It wants to send out scripts to certain predefined machines out there – which are already compromised by the virus to provide certain services. If the virus can't connect this these 'providers' it won't do any harm. So the mail scanner will assume that this attachment is ok, and the user will get it in his mailbox and click on it. Now the virus has direct Internet access, and sends these scripts to these compromised computers, and will do some sort of malware activity.

How should we attack these problems? Making it configurable that output from the simulated computers can pass out on the Internet, everything or only certain actions? We can of course fake any IP address, saying thanks for the package and smile – but the virus could want some services done. We could also scan these scripts or probes on the fly, as they want to go out – and maybe some generic detection scripts will detect them. In this case traditional AV techniques could be applied inside the simulated computers.

SIMULATING DYNAMIC LINKABLE LIBRARIES (DLL'S)

Many DLLs must be present on the simulated hard disk to make viruses believe they are running on a real system. Let's take a look at one of them, ADVAPI32.DLL. This library is used to give a ring3 interface to the Registry. It provides several ring3 level APIs to whoever wants them. How is this implemented in the sandbox?

All the libraries are written in Assembler, and they are linked as regular DLLs and stored in our definition file. They might import functions from other DLLs (like KERNEL32.DLL), in which case the DLL loader must resolve these imports. They export the functions we think are necessary and can be dynamically updated using our incremental definition file update mechanism.

Each API implemented in ADVAPI32 has the ability to:

- Send the request onwards to another destination (VMM, IFSMgr, other DLLs etc.). The VMM exports numerous functions to deal with Registry keys, and ADVAPI32 is just a ring3 'wrapper' towards the ring0 VMM interface.
- Just ignore and return (signal ok or error code).
- Handle the API (request) – perform some action (e.g. remember some values in case someone asks for it later).

The 'Registry' itself is located in 'ring0', and only available through VMM APIs. If there are other libraries also exporting functions to deal with Registry keys, we need to make a wrapper around ADVAPI32 to use the same functions – we don't re-invent the wheel unless we have to. The same goes for all the versions of WinSock using different libraries. ADVAPI32 is also offering the APIs to deal with tokens and SIDs.

WHAT IS MASS MAILING?

Many viruses carry the suffix @mm, meaning mass mailer. What does it mean? It means that the worm or virus will spread to other systems using mail – on a large scale. It could also spread using other techniques. They can operate on many levels, communicating with WINSOCK directly, email applications or filtering WINSOCK traffic. To create a suitable environment for these viruses to 'spread', these options must be left open so the virus can 'exploit' them. Detecting mass mailers based directly on replication means that you must simulate 100% the communication between the computer and the SMTP server responsible of sending the mail. You might also need to simulate email applications, like *Microsoft Outlook*.

To make a better understanding of how the simulated computer is able to simulate (through emulation) the scenario of mass mailing, I've included some case scenarios.

CASE SCENARIO: W32/SHEER.A@MM

This worm appeared during Christmas 2001. It is a tiny worm, written in Assembler. It uses functions from several Win32 libraries. It will attempt to download the file 'list.txt' from the web (http://banners.interfree.it/list.txt). This file was removed quickly after the worm was discovered, so the worm didn't pose any real threat. This file contains the email to send.

To make the worm believe it is executing in a proper computer, you need to write your own libraries so the PE loader can locate the entry points to simulate the executed functions within the simulated operating system. The code present in these libraries must behave as it would in the 'real world' – answer with sensible data and interpret the parameters given. The following libraries and APIs must be written and be available to make the virus work:

ADVAPI32.DLL	RegOpenKeyExA, RegQueryValueExA, RegCloseKey
KERNEL32.DLL	CloseHandle, CreateFileA, CreateFileMappingA, ExitProcess, GetModuleHandleA, GetProcAddress, GlobalAlloc, GlobalFree, GlobalLock, GlobalUnlock, MapViewOfFile, Sleep, lstrcatA
WININET.DLL	InternetCloseHandle, InternetGetConnectedState, InternetOpenA, InternetOpenUrlA, InternetReadFile
WSOCK32.DLL	closesocket(3), connect(4), htons(9), ioctlsocket(12), recv(16), send(19), socket(23), gethostbyname(52), WSAGetLastError(111), WSAStartup(115), WSACleanup(116)

The imports from WSOCK32 are done using ordinals, so the PE loader in the simulated KERNEL32 must also be able to import these using ordinals instead of names.

The fact that the virus wants to access the web to fetch the file generates another problem. If you enable the option to let the scanner download these files, you will detect the original file as a worm. If not, you cannot detect it as anything other than something trying to access a file on the web. Is that enough to be flagged as viral? No. That can be another option – we only allow these sets of applications to access the Internet (or none), but then we are moving over towards personal firewalls or behaviour blocking.

If this were a new virus that no one had ever seen, the file would probably have been online. So, for the paper, let's assume that the worm can access this file.

There is no harm in inserting data or code into the sandbox – provided that you never remove anything from it. As long as the code or data is inside the sandbox, it cannot touch the real computer, and any 'harm' the code could inflict, would be affected to the simulated computer and its simulated LAN.

To understand how a simulated computer can detect this virus, I need to explain in detail how the virus works. You will see for each API that our simulated APIs will perform the same actions the virus expects – if the computer were connected to the Internet. But nothing would leave the system or touch the real system.

The virus asks the base address of KERNEL32.DLL by calling *GetModuleFileHandle*. It then tries to get the API *RegisterServiceProcess* using *GetProcAddress*. This API isn't available on the *Windows 9x* series, so the virus would crash there.

Since my *Windows* clone is neither of the real versions, my KERNEL32 has support for this. Next, the virus will try to open the key 'Software\Microsoft\Internet Account Manager' of HKEY_CURRENT_USER. From this key, the virus tries to read the 'Default Mail Account'. The virus will construct a string looking like 'Software\Microsoft\Internet Account Manager\Accounts\{Current account}' and open this Registry key. The sub key 'SMTP server' will be retrieved and stored. Also the 'SMTP Email address' is retrieved and stored. It will then open another key, 'HKEY_CURRENT_USER\Software\Microsoft\WAB\WAB4\Wab File Name'. Now the virus knows where a Windows Address Book is stored.

What's interesting at this point is whether the dummy addresses specified in the WAB now suddenly will get messages.

The virus will now open this filename containing all the addresses using KERNEL32's *CreateFileA*. It will create a memory map of it using *CreateFileMap* and *MapViewOfFile*. Now it starts up the WinSock code by issuing a *WSAStartup*. The virus calls an API from WININET.DLL called *InternetGetConnectedState*. If this returns an error code (meaning NOT connected to the Internet), the virus sleeps for 0xEA60 (60,000) ms and tries again. When the computer gets connected to the Internet, the API will return ok, and the virus continues.

W32/Sheer.A@mm will call another WININET.DLL export, *InternetOpenA*. If this API fails, the virus will fall back to sleep for 60,000 ms and issues new calls to the *InternetGetConnectedState*. When connected, the virus will attempt to download 'http://banners.interfree.it/list.txt' from the Internet. If it doesn't succeed, the virus goes back to sleeping mode once again.

When the connection can be made, the virus allocates some memory using *GlobalAlloc* from KERNEL32, and it will also issue *a GlobalLock* on the memory. Next step is to download the file, using the WININET's API *InternetReadFile*.

The virus will now call a routine to parse the WAB file, fetching addresses and making WinSock commands to send the mail towards the configured SMTP server.

The virus will call a WSOCK32 export *socket*, which will return a socket descriptor. It will convert the port number (25) using the *htons* API. Next the virus issues a *gethostbyname* with reference to the active SMTP server. It should return a pointer to a structure giving the address of the SMTP server. It will identify the pointer to the address list, which is a NULL terminated list of addresses to the host. The virus will do another WSOCK32 API; *connect*, towards the first entry of this list. If successful, the virus takes a short break. After the rest, the virus sends the string 'HELO data.com' to the SMTP server using WSOCK32's *send* API. It pauses again before it continues by sending the string 'MAIL FROM: <' + the SMTP email address configured in Registry + '>'. It will pause again. Now it will send 'RCPT TO: <' + the mail address found in the WAB + '>'. Again it pauses. Now it will send 'DATA' and takes yet another pause. The next transmission is 'From:' + the email address found in the SMTP email address in Registry + linefeed. Pauses. The virus will then send the string 'To:' + the email address found in the WAB file + linefeed, and takes another pause. It then calculates the length of the downloaded file (list.txt) using a *strlen*, and sends the entire downloaded file. It will then close up. After another pause, it issues an *ioctlsocket* towards WSOCK32.DLL to control the mode of the socket. If the function returns ok, the virus executes the *recv* API from WSOCK32.DLL, receiving data back from the SMTP server into 0x403000, and zero terminates this data buffer. It then closes the socket, using *closesocket*. Then it runs the same procedure for every address in the WAB file.

I've appended a flow chart on how the virus works in Appendix B. Flow chart of W32/Sheer.A@mm.

To build the sandbox, and the APIs around it to make the virus believe it's sending mails is not a big task for this virus. The 'Internet' itself can actually be 'installed' in WSOCK32.DLL and a callback function, and all logic to 'preserve' the email towards the fake SMTP server can be done here. It doesn't have to leave, and we can extract the full mail body, and see whether the program attempted to send a mail to the email addresses stored on the simulated hard-disk. If these again contain an executable attachment, it's quite obvious that we're dealing with a mass-mailing worm. We can even put this attachment on a 'clean' simulated computer and examine the behaviour.

When scanning this worm, using the simulated computer, this is how the output to the user *could* look, assuming the callback function to let the engine 'fetch' the file, and assuming this was a new virus and the file was present:

C:\TEMP\JAVASCRIPT.EXE – infected with W32/Worm@mm.generic

- From: me@playground.no
- To: you@playground.no
- Using: SMTP.playground.no
- Subject: Fw: Scherzo!
- Body: 'Con questa mail ti e stata spedita la FortUna; non la'…
- Attachment: javascript.exe (6,656 bytes)
- Uses 'Incorrect MIME' security hole

CASE SCENARIO: W32/KLEZ.H

KERNEL32.DLL	GetComputerNameA, IsDBCSLeadByte, WriteFile, ReadFile, GetTempFileNameA, MultiByteToWideChar, CopyFileA, SetFileAttributesA, FindClose, FindNextFileA, FindFirstFileA, SetEndOfFile, LocalAlloc, GetTempPathA, DeleteFileA, WideCharToMultiByte, CreateProcessA, GetSystemDirectoryA, GetCurrentProcess, SystemTimeToFileTime, GetSystemTime, GetVersionExA, GetVersion, WaitForSingleObject, GetCommandLineA, ExpandEnvironmentStringsA, GetDriveTypeA, CreateThread, GetCurrentProcessId, GetLocalTime, LocalFree, GetLastError, SetFilePointer, GetFileTime, GetFileSize, FreeLibrary, LoadLibraryA, UnmapViewOfFile, CreateFileA, Process32First, CreateFileMappingA, MapViewOfFile, CreateToolhelp32Snapshot, Process32Next, GetModuleFileNameA, ReadProcessMemory, Module32First, OpenProcess, CloseHandle, TerminateProcess, Sleep, SetFileTime, GetTickCount, GetProcAddress, LCMapStringW, LCMapStringA, FlushFileBuffers, SetStdHandle, HeapReAlloc, VirtualAlloc, GetStringTypeW, GetStringTypeA, RtlUnwind, VirtualFree, HeapCreate, HeapDestroy, GetFileType, GetStdHandle, GetModuleHandleA, GetStartupInfoA, ExitProcess, GetCPInfo, GetACP, GetOEMCP, SetHandleCount, HeapFree, HeapAlloc, UnhandledExceptionFilter, FreeEnvironmentStringsA, FreeEnvironmentStringsW, GetEnvironmentStrings, GetEnvironmentStringsW
ADVAPI32.DLL	OpenSCManagerA, StartServiceCtrlDispatcherA, LookupPrivilegeValueA, AdjustTokenPrivileges, RegSetValueExA, RegQueryValueExA, RegCreateKeyA, RegConnectRegistryA, OpenProcessToken, StartServiceA, AllocateAndInitializeSid, EqualSid, GetTokenInformation, RegisterServiceCtrlHandlerA, OpenServiceA, FreeSid, CloseServiceHandle, RegEnumValueA, CreateServiceA, RegOpenKeyA, RegEnumKeyA, RegDeleteValueA, SetServiceStatus, RegCloseKey
WS2_32.DLL	closesocket(3), connect(4), htons(9), recv(16), send(19), socket(23), gethostbyname(52), WSAGetLastError(111), WSAStartup(115), WSACleanup(116)
MPR.DLL	WnetOpenEnumA, WnetEnumResourceA, WnetCloseEnum

Comparing the imports from WS2_32.DLL and W32/Sheer.A@mm's imports from WSOCK32.DLL don't differ much. W32/Sheer.A@mm imports one more function (*ioctlsocket(12)*), but the remaining ten functions are 'identical'. You can also see that W32/Klez.H@mm is a more complex virus just by looking at the import tables. These functions must be provided and return sensible data filled in structures to make W32/Klez.H@mm believe it's running inside a real computer, connected to a LAN. This is the biggest job of building this kind of system.

Parsing the runtime code is another topic (see Appendix A: Tracing through the runtime of a *Microsoft* C application). Assume that the EIP is at the WinMain of the Win32 application. All runtime has been emulated, and all initialization has been done.

This virus is using multiple threads, so every bit of thread simulation is necessary to emulate/simulate the execution of this virus. The WinMain is located at 0x4073e2, and starts by calling the *WSAStartup* from WS2_32.DLL (WinSock2). It then starts a series of calls, including one to decrypt the encrypted data part inside the virus. The virus will then check whether it has administrator rights or not. It will check what platform it's running on, and launch the threads, using *CreateThread*. The virus now executes in several threads, all performing specific tasks. One enumerates all the processes and checks for presence of anti-virus, and kills them. Each drive will have its own thread infecting files, creating the hardly 'compressed' data files. Another will enumerate the network for shares, and constantly bombard the shares with worm files. Another thread is in 'charge' of the mass mailing, while another thread drops the W32/Elkern.C virus. Going into details on this virus would make this paper too large and uninteresting for the better part of the audience, so I won't do that. Many AV vendors have excellent virus descriptions of this virus (see http://www.norman.no/virus_info/w32_klez_g_mm.shtml). The purpose of this discussion is to get an idea of how many APIs must be implemented and how accurate they must be so viruses can't figure out if they are running simulated or not.

As an example, let's look at thread 3. This thread is 'responsible' for spreading through network resources. The virus starts by calling *WNetOpenEnumA*, and export of MPR.DLL. This API starts an enumeration of network resources or existing connections. It specifies the resources to be enumerated to be RESOURCETYPE_DISK. It also specifies 'all connectable resources' and 'all container resources'. If this API returns an error, the virus will return and sleep for a while. If not, the API *WnetEnumResourceA* is called. This API uses the handle provided by *WNetOpenEnumA*, and will fill out a buffer with sequential NETRESOURCE structures. Again, the virus stops enumerating if the count of available resources is 0.

If the resource found is a container, the virus will start enumerating the network resources within this container, using the *WnetOpenEnumA* and *WnetEnumResourceA*. If it finds another container, it will call itself recursively.

When it has found a resource that isn't a container – but derives from one – it will split the path of the resource into pathname and machine name. If not running under *Windows 9x* and the API *NetShareAdd* is available, it will retrieve its own computer name, using the API *GetComputerNameA*. It will match the name of the resource found against the name returned. If it's not identical, the virus will try to open a share towards this resource, filling in a SHAREINFO structure and calling the API *NetShareAdd*.

The spreading mechanism from here is the same – coming from a container or not. The virus builds its remote filename, copies the executable file over (using *CopyFileA*). If running on a *Windows NT* clone with enough privileges (it must have domain administrator rights) on the remote system, it will try to launch the copied file as a service (using *CreateServiceA*) on the remote computer. If the API *NetShareGetInfo* is available and still running under *NT*, the virus connects to the Registry of the remote computer, and attempts to create a key to launch the virus locally. It will also try to copy a .RAR file to the remote resource. The virus is multibyte character aware; so all APIs dealing with multibyte (*MultiByteToWideChar*) must be simulated as well.

To make the virus believe it's running on a true computer, communicating with remote systems means writing a lot of APIs and having a network to enumerate. The virus also performs some privilege checks, creates remote services and Registry keys – all of this must be simulated. The easiest solution is to simulate that you're a *Windows 9x* box – then the virus still spreads, but doesn't apply that large amount of APIs and checks.

SPREADING THE OLD WAY

Many viruses don't have mass mailing capabilities, and spread using regular file replication. There are mainly two different approaches used: creating memory mapped views of files and the old-fashioned file handle operations (seek/read/write etc.). When dealing with memory mapped files, the 'victim' file is paged into memory space, but the pages are not loaded before an opcode requests the specific memory. When the map is closed, data is committed to disk if opened for write. In order to simulate this, page not fault and opcode restart must be implemented, or the memory mapping API's must load the entire file when dealing with these API's. Neither of these methods poses any problems for the simulated OS.

In my presentation last year I demonstrated how W95/Zerg.3469 replicated inside the simulated computer. This virus hooks the IFSMgr and will infect files on close. It also features stealth capabilities. Viruses that mainly do file replication lay on ring3 (the application level) to be compatible with *Windows NT*/*W2K*/*XP*. KERNEL32 export numerous functions to find files and manipulate them.

RUNTIME LIBRARIES

Many worms on the Win32 platform are written using so called high-level languages. It means that it simplifies the process of creating it, using a set of library functions provided by an SDK. The 'C' language is also a high level language, which provides APIs to do various tasks. Many high-level languages require the presence of a library DLL, like Visual Basic. This means that the APIs used by the application isn't statically linked together with the application, but needs to be resolved at runtime. Even programs written in 'Visual C' require the presence of MSVCRT.DLL.

To let these applications 'run' in the simulated computer, you need to have these libraries available on the simulated hard-disk. These libraries tend to be large and export hundreds of functions. As an example, take the library MSVCRT.DLL of 00-07-00. It requires the presence of 144 functions in the KERNEL32.DLL. The MSVCRT.DLL itself exports 759 functions. Then you can probably figure out the work involved writing these libraries.

Why not copy the real ones to the simulated hard disk? Well, there are several reasons:

1. There are numerous versions of them – which one should be used? All of them?

2. They could be infected – then what is infected, the DLL of the application using it?
3. They would require that we have all these libraries on the system to detect the virus (e.g. MSVCRT.DLL on Netware?). Detection could also differ depending on which DLLs present on one system compared to another.

Why not store all these real production libraries in the anti-virus definition file? It would induce problems related to copyright as well as size. I ended up writing the DLLs for *Windows* that I needed, and I will probably have to continue to do that. They are not complete, but they have 'enough' support to let the virus believe it is connected to the real one.

What is actually used by viruses in these runtime libraries? Obviously, the startup code uses some APIs to perform the necessary initialization. To pinpoint what APIs to 'simulate', you need to trace through these applications to verify which APIs are actually used.

Applications written using Visual Basic use more runtime libraries than e.g. Visual C. You can compile the applications in different modes (native or pcode-oriented), and different versions use different libraries. The typical Visual Basic executable starts by calling a function within the runtime library, giving as a parameter a structure describing to the runtime how to proceed. This will be very hard to simulate, since you would need to understand the entire format of the Visual Basic executables, and write your own runtime libraries. For now, these applications won't run inside the simulated computer due to the lack of runtime library support.

ODD TECHNIQUES

A long time ago, we believed that viruses using their own DLLs for supporting APIs would be created. Having a DLL exporting e.g. the polymorphic engine, the decryption code or APIs to perform replication. How would this work in the sandbox?

A virus would use e.g. the *LoadLibraryExA* to load the relevant library. Only the initial application is 'located' inside the simulated hard disk, so any 'special' DLL it wants to use, would not be accessible. However, during the *LoadLibraryExA*, you could 'install' the library from the 'real world' – on access. You would only do this if the library wasn't known, and it's present in the same directory. Now, this points to another interesting question: Which file is 'infected'? What if the DLL is the virus (say the virus has overwritten a DLL file on your system, and replaced some exports) – the application you are testing for behaviour is clean – it just happens to call this DLL?

This is the trouble with importing code from a 'real' system. If you want to make these kinds of libraries available to applications, the risk is that you false alarm on innocent executables that just happen to use an infected library you're unaware of.

What about viruses that replace the original WSOCK32.DLL with their own? They usually rename the existing WSOCK32.DLL (for example to BORING.DLL), and copy their viral to WSOCK32.DLL. When an application is to use this DLL, the library is loaded using KERNEL32s *LoadLibraryA*. The viral library is then loaded, which in turn loads the original since it's just a filter. It will pass on requests it doesn't care about to the 'real' one, and maybe attach data on the packages going through if it wants to. The original WSOCK32.DLL is still the

DLL providing the actual APIs, so this scenario will also work in the simulated computer. Since an installed ‘library’ like this would be dormant waiting for someone to actually send a mail, a goat file can send an empty mail – and the SMTP server can verify that it actually received the mail unchanged. If it carries an attachment, this can be inserted into another clean simulated computer.

ANTI EMULATION OR DEBUGGING CODE

What is anti-emulation code? It is code that the virus authors think will behave differently when emulated compared to execution in ‘real life’. Anti-debugging code is code that should detect the presence of debuggers, single stepping etc. All in all, it’s a challenge to make the emulation as perfect as possible.

In the old MS-DOS days, simple tricks were deployed to detect the presence of single stepping pushing a value on the stack, popping it – and verifying that it still existed on the stack (no single step interrupt in-between). Playing around with the prefetch-queue (PFQ) was also a ‘trick’ used to differentiate between actually running under a debugger or not. The code inserted a CD 20 in the memory space of the next instruction, but since this memory was already cached by the PFQ, there would be a difference in executing under a debugger (INT1 single step) and running for ‘real’. Since there was a change in the processor handling of the PFQ, this ‘trick’ no longer applies.

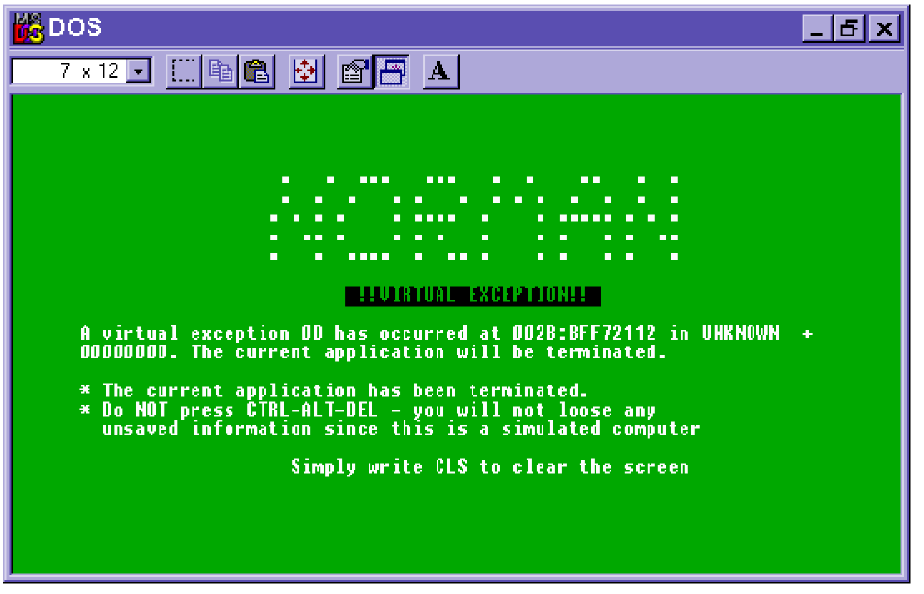
Writing to a memory address that hopefully was absent during emulation is another trick employed. By writing to a weird address – reading it and comparing it – virus authors hoped to be able to distinguish between emulation and real execution. Most AV scanners support the 4 GB virtual memory space and the translation of descriptors, providing the same support during emulation as for real execution. We even provide virtual exceptions for ‘page not found’, so our ‘*Windows*’ page fault handler will ‘load’ the page from the executable as well – on demand. For instance, the instructions ‘**mov eax,[430000h]**’ if the page located in the linear memory space of the current descriptor is absent – an interrupt will be issued. This interrupt will examine the situation, and figure out that this page actually reside in the Win32 image we’re ‘executing’ – and load it. It will then restart the opcode. From the opcode’s point of view nothing strange happened – it doesn’t know. Special care has to be implemented in opcodes like interrupts and jumps switching tasks with regards to ‘page not found’ so the emulator doesn’t recursively apply exceptions in exceptions in a never-ending loop.

Another anti-emulation trick is to call imports from various libraries in the decrypt loop of viruses. The virus authors then hope that the AV would give up decrypting when the call enters e.g. KERNEL32.DLL. One of the first Trojans to use this technique appeared several years ago.

Win32 PE executables can set up a so-called SEH (Structured Exception Handler). Most runtime code, and several viruses use this method to trap errors, but it can also be used to intentionally cause a change in the execution flow. Viruses insert their SEH and create a fault condition, causing the exception handler to be run instead. The first virus really in ‘in-the-wild’ installing multiple exception handlers was W32/Magistr.A to confuse AV emulators.

Our emulator is capable of emulating most instructions, even MMX. It is able to emulate properly e.g. single step mode generating the INT 1 as it would in the real world (e.g.

Dark_Paranoia). It also emulates task switching, switching between real and protected mode, and would generate most of the exceptions a regular CPU would on the same scenarios. Our '*Windows 9x*' clone even has a blue screen handler installed, and will print the proper values on the simulated screen. I have changed the screen colour to green, since people tend to 'panic' when a blue screen appears.



COMPRESSED EXECUTABLES

A compressed application using a third-party application like Petite, ASPack etc, causes problems for some scanner engines. The application itself, when run, will decompress itself, and start the original application. However, anti-virus software must decompress the outer layer(s) in order to view the inside.

Placing an executable of this type into the simulated hard disk doesn't make any difference. The loader won't see any difference, and the APIs needed by the decompressor will be resolved at runtime. When the applications has decompressed itself, it will load the other required DLLs and locate the necessary imports, and start the 'original' application. This is exactly the same how it's handled in the simulated computer as well. Speed is the only problem, since it requires emulation of millions of instructions to decompress large compressed executables. With an average speed of one million instructions emulated per second (on an average desktop – PIII 700 MHz), this isn't the preferred way of decompressing these executables, but it can be done if the engine doesn't support the format. If an external handler knows how to decompress the executable, they should do this before the decompressed executable is inserted into the sandbox – to save CPU cycles.

SECURITY BREACHES

What could be reported as security breaches from this simulated computer? That an application installs something that listens to the keyboard? That files containing passwords are being exported, or drives are being shared out – openly with full rights? That guest accounts are being created with administrator rights? Maybe the act of opening ports on the computer? The possibilities are very interesting, but we haven't implemented any support for this yet.

UPDATE THE SIMULATED COMPUTER AND THE LAN

All the software residing on the simulated hard disk is located in our signature data file. This signature file can be updated using incremental signature files. So if we make a change e.g. in an API in ADVAPI32 or add a new library (e.g. OLE2.DLL), we can add it and the users don't have to download a complete signature data file – just the incremental. Even the BIOS is looked upon as a software module, residing in read only memory.

The simulated hard disk is dynamical, and adjusts after the size of the software modules put into it, and also the sample we're investigating. The number of cylinders, heads and sectors will change depending on the total size required.

IO port updates require (for the moment) an engine update, but hopefully the software modules, or some sort of intermediate language to be interpreted by the engine, will also cover this.

Network configuration, like machine names, IP addresses etc. will also be stored in the signature data file, letting us change the configuration, number of machines, etc. on the fly. The SMTP server answering to the WinSock requests is also just a software module, and can be extended as the future reveals new threats. The configuration of shares open between the simulated computers will also be configurable, providing us the possibility of opening new shares and changing access rights as we see fit.

APPENDIX A: TRACING THROUGH THE RUNTIME OF A MICROSOFT C APPLICATION

If a virus is written using a high-level language, the simulated computer must first emulate the runtime code. This runtime sets everything up (device handles, environment, localization (MBCS) arrays etc). As an example, I've gone through the entire C runtime used in the W32/Klez.H virus – to see what APIs and emulation is necessary to let the runtime initialize itself, and call the WinMain – where the viral code begins.

The pseudo code of this runtime is provided below. It was made running under English *Windows 98 SE*, code page 1252. I used NuMega's SoftIce and IDA (Interactive Disassembler) from Data Rescue to gather the information.

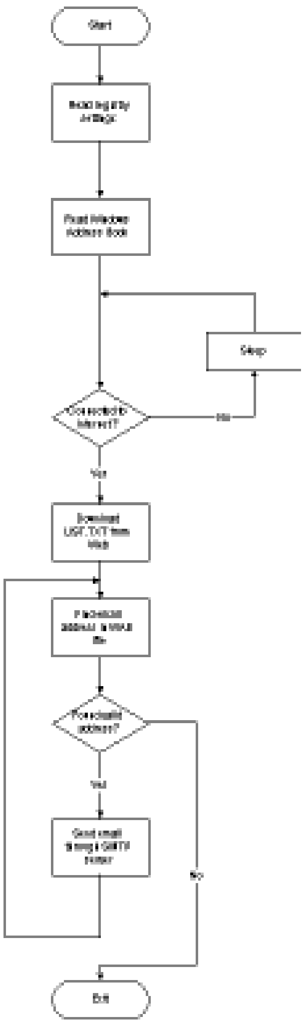
```
Install exception handler (FS:0)
Call KERNEL32!GetVersion: returns eax=0xC0000A04
Call KERNEL32!HeapCreate()
Call KERNEL32!HeapAlloc(hHeap,0,0x140)
Call _malloc() -> _nh_malloc -> KERNEL32!HeapAlloc()
```

loop preparing preparing the allocated memory with FFFFFFFF,00000A00
Call KERNEL32!GetStartupInfoA(&STARTUPINFO)
Checks wheter StartupInfo.cbReserved2 == 0 - it is
Call KERNEL32!GetStdHandle(0xFFFFFFF6): returns FFFFFFFF

```
Call KERNEL32!GetStdHandle(0xFFFFFFF5): returns FFFFFFFF
Call KERNEL32!GetStdHandle(0xFFFFFFF4): returns FFFFFFFF
Call KERNEL32!SetHandleCount(0x20): returns 20h
Call KERNEL32!GetCommandLineA(): returns address of command line
Call _crtGetEnvironmentStringsA
    Call KERNEL32!GetEnvironmentStringsW(): returns 0
    Call KERNEL32!GetEnvrionmentStringsA(): returns ok (0x5B0020)
    Calculate size of environment block + 1
    Call malloc(buffer to fit environment block):
    Call memcpy to copy environment block

Call KERNEL32!FreeEnvironmentStringsA
Call __setargv
    Call __initmbctable -> sets flag so it doesn't reinit itself
        Call __setmbcp(FD)
            Call KERNEL32!GetSystemCP: returns 0x4E4
        Call KERNEL32!GetCPInfo(): returns 1
        Call KERNEL32!GetCPInfo(): returns 1
        Intrinsic memset(0x100 of stack memory)
        AL = ebp.CPInfo.LeadByte = 0
        Call __crtGetStringTypeA(1)
            Install Exception handler(FS:0)
            Call GetStringTypeW(): returns 0
            Call GetStringTypeA(): returns 1
            Call GetStringTypeA(): returns 1
            Unhook exception handler
        Call __crtLCMapStringA
            Install Exception handler(FS:0)
            Call LCMapStringW(): returns 0
            Call LCMapStringA(): returns 1
            Loop: Sets up a multi byte array
            Call LCMapStringA(): returns 100
            Unhook exception handler
        Call __crtLCMapStringA (described above)
    Call KERNEL32!GetModuleFileNameA(): returns length
        Call _parse_cmdline
        Call _malloc()
        Call _parse_cmdline
Call _setenvp
    LOOP through all strings in environment block
        Count strings that doesn't start with '='
        Call _strlen(EnvironmentString)
    LOOP through all strings in environment block
        Call _strlen(EnvironmentString)
        IF string does not start with '='
        Call _malloc(LengthOfString)
            Call _strcpy(allocatedBuffer, EnvironmentString)
```

```
                ENDIF
Call _free (pEnvironment)
Call __cinit
        Call __initterm: table of init function pointer to call
                Call ___initstdio()
                        Call _calloc(200,4) -> KERNEL32!HeapAlloc
                        Builds a function pointer list (0x20 in between)
        Call __initterm: table of init function pointer to call
                <no handlers in list>
Call KERNEL32!GetStartupInfoA(): returns pointer
        Call _wincmdln
                Parses the command line

EAX = StartupInfo.wShowWindows = 1

Call KERNEL32!GetModuleHandleA()
Call _WinMain@16(hInstance, hPrevInst, lpszCmdLine, CmdShow)
```

APPENDIX B: FLOW CHART OF W32/SHEER.A



Windows Sandbox suffers from an annoying issue where it just won't connect to the internet. Use these troubleshooting tips to resolve the problem immediately.Â The Windows Sandbox lets you do just about anything. You can try out sketchy-looking programs and websites or mess around the operating system settings without worrying about breaking things up. Download Sandboxie for Windows to browse the web securely and avoid unwanted changes to your system. Sandboxie has had 1 update within the past 6 months.Â Browse the web securely and avoid unwanted changes to your system. Last updated on 04/06/20. Sandboxie is an open-source sandboxing program for Microsoft Windows. Sandboxie creates an isolated operating environment in which applications can be run or installed without permanently modifying the local system. This virtual environment allows for controlled testing of untrusted programs and web surfing. After various ownership transitions (Sophos acquired Invincea which acquired Sandboxie from the original author Ronen Tzur), Sophos eventually stated they would no longer be involved in the