SANDBOX TECHNOLOGY INSIDE AV SCANNERS

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

How can we reliably tell that an executable contains an unknown virus? Are we guessing based on fractions of known viral code? Does it 'look' like something we already know, or does the composition of opcodes 'look' like they can perform viral behaviour? Many scanners use different techniques to discover new variants, or even totally new viruses. Searching for the unknown has always been prone to false alarms. I will present how Norman Virus Control (NVC) v5 is able to detect known and unknown viruses with a very low risk of false positives, if any. I will describe how it is done, including the hit rates and experience from the real world. Further, I will present ideas on how it will be expanded in functionality and how to detect more classes of binary viruses. A demonstration of the system will be given. This method does not care whether a virus is encrypted, polymorphic, metamorphic etc. It relies on the basics of a virus: it must replicate!

WHAT IS A SANDBOX?

Where do you usually find a sandbox? In the kids' playground. They use it to play around in, building and tearing down structures.

A sandbox inside a scanner engine is also a playground – for computer files. Executable files can be 'test-driven', and the results can be analysed to see whether it is a good thing or not to run it on your system. Not 'scanning' for malware, but running it to see what it does, rather than what it looks like it might do.

There are many definitions of a virus. For this paper, I shall use the following definition:

A computer virus is a computer program, defined through its behaviour. It will transfer code/data to other computer files. When this other computer file is given control, the virus code is somehow activated, trying to infect other computer files. This process is called replication. For a computer program to be called 'viral', it must be able to perform this task recursively.

SANDBOXING USING EMULATION

My sandbox is a virtual world where everything is simulated. It is powered by an emulator, and together they let possible virus-infected binary executables 'run' as they would on a real system. When execution stops, the sandbox is analysed for changes.

However, viruses can detect that they run inside a simulated world. Even though I try to cover most aspects, there will always be an API or service I don't provide which viruses can test. I only implement a subset of DOS/*Windows* – just enough to run viruses and their hosts. Many viruses are not stable programs. They contain bugs, and often causes your real CPU to hang. This is also a problem for simulation. When does it hang and when is it just a really slow, huge virus?

Since nothing runs on your system, except an emulator, it is safe. Everything else is virtualized. Nothing is stored on your hard disk, even if the sample you are testing wants to. Since sandboxing with the purpose of detecting viruses includes 'booting' the 'computer', having a full-blown installation of e.g. *Windows 98 SE* would be too slow and inefficient. My environment is simulated and tailor made – it is written by me. It takes a lot of time developing the operating systems and testing them so that they match the real ones to a certain degree. You can 'apply' any tailor-made operating system to inspect viral/malicious behaviour of a sample. The entire computer is virtualized. A new BIOS, CPU feature, OS fix, etc can be virtually applied, and several worlds can be simulated in turns. Everything is controlled from our definition file.

It runs inside the scanner engine on any computer/OS we're already supporting, and will detect the same viruses on the different platforms, even on access if you like.

SANDBOXING USING A VIRTUAL MACHINE

You could also build a sandbox by creating a VM (Virtual Machine). The idea is to tie all exits,

so the executable you are examining can't escape. In my opinion, it's not safe. There will always be 'another' exploit of the CPU, some weird interrupt/exception/fault etc. that will allow malicious code to escape a VM, and touch your real system.

We've seen it before with other attempts to 'fence' malware (e.g. INT 1 tracing). A VM is too specific to the environment you are running, and cannot be a part of a generic scanner engine. A VM doesn't virtualize the computer your running on. The hardware, CPU and BIOS are equal to the one already running on the system. Using a VM, you cannot test suspicious files for other platforms (e.g. *Linux*). Your system is not necessarily clean – you use what you have. However, executing inside a VM is faster. Everything happens in real time. The environment is already there, you don't have to 'create' it. You use a copy of the machine as it is, which saves development.

BACKGROUND

First, it started as a test project to detect viruses like the ACG series. These viruses are metamorphic, and do not use any encryption to hide themselves. The virus body mutates from infection to infection, and there is no layer to decrypt to find any 'static' data to scan. Well, during this process it turned out that both ACG.A and ACG.B have an encrypted engine inside, so you can identify it correctly. The project, however, continued. ACG is complex, and it uses a lot of instructions to replicate to another goat file. It was a initial goal to make both ACG.A and ACG.B replicate recursively inside the scanner engine, which causes no other problem than time. On average, ACG.A uses 9,576,196 instructions per sample to infect a file. For the record, the highest number encountered was 31,963,016 instructions and the lowest was 5,724,066 instructions to replicate one sample, once. An initial goal was to make this system detect heuristically anything binary getting into the WildList. This is still a goal, but we're not there yet.

Another reason for implementing this was to create a new method of heuristics for *Windows* viruses. *Windows* PE executables are usually bigger than DOS executables, and they normally require input from other libraries in order to run. These applications also run under protected-mode rather than real-mode, giving a 'false' sense of security. Protected-mode also gives a challenge to your emulator.

In the early days of NVC 5, this technology was only activated internally in the engine when we saw something interesting. Now, it must be enabled through the configuration editor, inside the common settings. It's not activated by default.

WHAT RESOURCES MUST BE SIMULATED?

What parts of a PC must you simulate to make a virus believe it's running on a real system?

Storage Device Fixed disks of any size or layout. Will be initialized with a file system

(logical drive C), including system and goat files prior to bootstrap

process.

IO manager Will handle all I/O requests, and will also have access to RAM. ROM Virtually Read-Only-Memory. Used to store BIOS code.

RAM	Initialized to a known value before bootstrap. Memory below 1 MB is
	paged linear, and memory above 1 MB is paged using linked lists.
User	We need a virtual user to be there to control mouse and keyboard
	activity. Sample values are pre-inserted to keyboard.
CPU	An emulator, that can emulate any instruction incl. Multi Media
	eXtentions (MMX), floating point unit (FPU) etc. Flag handling,
	interrupt/exception/fault handling is also done by the emulator.

What I/O ports must be 'simulated'? Viruses tend to use timers in particular to generate random values for them. Some viruses read and write sectors directly to the hard disk talking to the hard-disk controller. This must also be perfectly simulated if you want to catch all boot infectors. EGA/VGA/CRT ports are also used, but not by many viruses. These should return sensible data so legitimate executables can continue without problems.

Since everything is virtual, and time is critical, you don't want to stop after every instruction to see what's going on. You rather let the system tell you (poll) when something is happening, using reserved ports and values is one technique that has proven useful. Since the point is to make viruses replicate, and time is the most expensive resource, the system should try to stop as soon as possible if we're emulating a clean program. This is the hard part. Many optimization techniques can be applied to make things go faster and avoid doing certain tasks, but the more weird logic you put into it, the bigger the chance there is a virus out there that isn't detected because it's doing this — maybe even without knowing it.

An emulator takes care of the CPU functionality. You need a good memory manager to virtualize 4 GB of memory, without using too much of the real memory or huge overhead when accessing memory. The memory manager must also understand differences between linear and physical memory, and how descriptors works.

The virtual storage device can be a removable or a fixed media, and it can be partitioned into any logical size and file system. Today FAT16 is used as a default file system. One virtual storage device corresponds to one 'physical' disk connected. Inside each virtual storage device, you make partitions like you would on a regular system. There is no limit on how many 'physical' drives you can connect. Shared drives is something which will be implemented later.

Here is a sample layout of how virtual drive C: could look like:

FILENAME		SIZE	CLUSTER	TIME	DATE	
IO	.SYS	008160	0002	15:30:44	12.05.2000	archive
MSDOS	.SYS	000007	0018	15:30:44	12.05.2000	archive
WINDOWS		<dir></dir>	0019	15:30:44	12.05.2000	directory
FILEC	.COM	004000	0020	15:30:44	12.05.2000	archive
COMMAND	.COM	004575	0028	15:30:44	12.05.2000	archive
FILEA	.COM	001002	0037	15:30:44	12.05.2000	archive
FILEB	.COM	002002	0039	15:30:44	12.05.2000	archive
FILED	.COM	016384	0043	15:30:44	12.05.2000	archive
WIN	.COM	005249	0075	15:30:44	12.05.2000	archive
EDIT	.COM	001002	0086	15:30:44	12.05.2000	archive
FORMAT	.COM	002002	8800	15:30:44	12.05.2000	archive
FDISK	.COM	004000	0092	15:30:44	12.05.2000	archive
WINVER	.COM	016384	0100	15:30:44	12.05.2000	archive
GREP	.COM	007023	0132	15:30:44	12.05.2000	archive
MM	.COM	005328	0146	15:30:44	12.05.2000	archive
MEM	.EXE	001000	0157	15:30:44	12.05.2000	archive
XCOPY	.EXE	002000	0159	15:30:44	12.05.2000	archive

SETVER	.EXE	004000	0163	15:30:44	12.05.2000		archive
EMM386	.EXE	016920	0171	15:30:44	12.05.2000		archive
NU	.EXE	001000	0205	15:30:44	12.05.2000		archive
SBSCOPY	.EXE	002000	0207	15:30:44	12.05.2000		archive
MSD	.EXE	004000	0211	15:30:44	12.05.2000		archive
ED	.EXE	016920	0219	15:30:44	12.05.2000		archive
PMAP	.EXE	016920	0253	15:30:44	12.05.2000		archive
TOUCH	.EXE	010754	0287	15:30:44	12.05.2000		archive
SAMPLE	.EXE	002626	0309	15:30:44	12.05.2000		archive

Look at the SAMPLE.EXE. This is the SAMPLE FILE we're going to investigate. It's not really there. I don't want to copy the entire file from the real world over to the simulated world. The directory structure is filled in, and the FAT information is correct. The secret lies in the generic storage device. It knows that the sectors belonging to the clusters of this file are file-based. On an access to read/write these clusters, it will be fetched from the physical file. If someone attempts to write to the same clusters (or expand it), a read/write copy the modified clusters will be stored in memory. Same goes for all the GOAT files. They are READ-ONLY copies, coming from the definition file. Any attempt to modify them will generate a new cluster that is READ-WRITE, on the same cluster, only the logical link internally is changed.

It's up to the operating system load to load the SAMPLE FILE into memory. How it does it, is up to each operating system. It could depend on page faults, forcing pages to be loaded on real use (Protected Mode) or it might load the entire file at once (Real Mode). The operating system loader will also take care of base relocations and import/export functionality.

The system must be restored 100% every run. This means that all 'resources' used must be fresh – every time. There are tricks to make this faster (e.g. make a read-only copy in memory and some sort of paging into this one). Since the system 'boots' for every replication attempt, the virtual BIOS sets the basic environment, like interrupts, BIOS data segment etc. The BIOS will then ask the CMOS what device to boot from. Normally this will be the physical drive 0x80. The MBS recognizes that we have a partition (drive C) using a known file system, and will boot. IO.SYS will be loaded, which in my case holds my real-mode 'DOS'. The operating system will initialize itself, hooking various interrupts, creating various tables (CDS, SFT, MCB, DEVICE DRIVERS, etc).

It's essential that every structure you use matches the real world, because DOS is a very old and dissected platform, so many viruses go directly into structures/locations (e.g. SFT tables). All data modules are put into the definition file, so only the emulator is real code running inside the scanner engine. This makes the system very configurable, and you have the potential of having several worlds, and combinations of them.

```
8 x 12 A

NORMAN virtual BIOS v1.10

Miniature DOS 2.00 (C) Norman ASA 2001 - Written by Kurt Natvig INT 21h being traced

This file is infected with a virus!

Pre infection file size = 10,000

C:\FILEC.COM
```

Figure 1.

When done, it will try to spawn the default shell.

WHAT ABOUT THE USER?

What user? Since this system is non-user-interactive, we can't wait for a user input. A keyboard-chain exists, but it is only initialized with a fixed number of keystrokes. The keyboard ring can be read and written to. Mouse movements and other 'user' activity can be 'virtualized', but so far that hasn't been necessary. A monitor is 'connected', so viruses can write to the screen or use the video memory for whatever they like. In the AV-lab version of NVC, the VIDEO memory is redirected to another real MS-DOS-BOX so I can view the screen in a window – real time. My DOS is supporting some video APIs, and goes through my BIOS interrupt 0x10 to show things on screen. It's also a nice feature to show debug messages, or information about what a virus does; like the 'INT 21h being traced' message above. It's actually my INT 0x21 handler that 'sees' that someone is tracing it, and prints the message, using standard interrupt 0x21, service 0x09.

WHAT OPERATING SYSTEM TO START?

My shell, COMMAND.COM, checks to see the format of the SAMPLE FILE we're going to investigate, and will either start it right away (service 0x4B/int 0x21), or it will start my WIN.COM, which will start my version of the *Windows 9x* kernel, which is a non-GUI version.

My WIN.COM will set up all descriptor tables and enter protected-mode in ring 0. It will set up all necessary structures required to continue. It will eventually load two VxDs, VMM and IFS. So far I haven't had use for more, but it can be dynamically expanded. My shell program will be loaded as a ring 3 application, and the SAMPLE FILE will be executed using the KERNEL32!WinExec. All WIN32 APIs are implemented in regular Dynamic Link Libraries (DLL) files (stored in the definition file). Inside KERNEL32, each API perform the given task. Many of them do nothing, just return. WinExec will locate the file on the virtual storage device, fetch imports (if available), and give control to the program. Since I didn't want to implement too many ways of opening a file, reading a file etc., ring 3 calls eventually comes down to ring0. RING0_API's will generate IFS packages, and follow the chain. The last receiver is my IFSMgr which will perform the actual task by entering 'real mode', calling regular INT 0x21/0x13 from the DOS underneath my *Windows*.

An example of an open file request from an application:

```
CreateFileA ("C:\GOAT1.EXE", GENERIC_READ or GENERIC_WRITE,
.....);
_IFSMgr_RingO_FileIO (RO_OPENCREATE, "C:\GOAT1.EXE", ....,
ATTR_ARCHIVE);
FSD (C000111F,IFSFN_OPEN,3,IFSFH_RES_LOCAL,0,53835C);
INT 21/AX=3D02 : open existing file : C:\GOAT1.EXE, Handle 0005 |
CF=0
```

You might say there's no 32-bit file system here! Only 32-bit filters converting parameters and results. Regular file requests moves from ring3 → ring0 → real-mode and up again. Conversion

of different status bits and addresses takes the most time to write.

RINGO_ functions are, before they are used, just an interrupt 0xCD 0x20 xx xx xx xx, in which the following DWORD shows which VxD should handle the request with what function. My INT 0x20 handler will do the same as the real INT 0x20 handler, and convert the 0xCD 0x20 xx xx xx xx to an indirect CALL. (0xFF 0x15).

If we were allowed to put in the original *Microsoft Windows 98 SE*, the bootstrap process, loading vmm32.vxd, various DLLs, initializing PnP devices etc. would kill the emulator. It would take forever to 'boot', and the program would be useless outside the virus analysis lab. That's why I keep it as small as possible, but with the functionality executables, and viruses especially, I want and need in order to run.

Our binary definition file (which today is approx. 600 KB) holds all signatures, virus information, disinfection information, system files, goat files etc. It would grow to enormous proportions if you were to put in a complete installation of *DOS*, *Windows 98 SE*, *Linux* and other platforms you want to simulate. We would also have copyright issues.

Besides, we need more information about what's going on. We need 'spies' all over the system who can reveal what's happening. These spies are not located inside the 'original' software packages, so we need to write our own.

When the SAMPLE FILE has executed, and there still hasn't been any modification of binary executables (goats), the system must try to access files on the virtual drives. It will read and write the MBS. It will also do a find first/next, opening all *.COM and *.EXE files on the system using several opening methods. It will also try to execute them.

If nothing has happened, the driver will reboot the sandbox. The BIOS will evaluate if it's necessary to allow the sandbox to have another go. If no binary files were modified and none of the system's boot sectors were changed, there's no need. The emulation would go down the same path. If e.g. COMMAND.COM or the SBS/MBS was changed, the sandbox is allowed another go.

TWO-STEP REPLICATION

In order to know that we're dealing with a unknown virus, we need to replicate it. If it replicates, we still need to know whether the new infection still will infect. One-step replication is not enough to filter off false positives or to be sure your dealing with a real-life virus. You could be dealing with an executable that compresses files or marks them for any weird reason. After determining what files were modified in the first run, the system reinitializes to a clean state and these files (which used to be clean) are copied into the 'sandbox', one by one. If other 'clean' executable files now become 'dirty', we're dealing with a virus – for sure. It's moved from the original sample to a file I've verified couldn't do viral behaviour. It then moved out of this file into other clean files.

I don't have to look at the sample to determine whether it's infectious, I've verified that by 'running' it. It doesn't matter what kind of virus it's in there. Whether they are plain simple, heavily encrypted or metamorphic, they still make previously clean files infectious.

Companion viruses don't modify the existing file, but create a new, normally hidden, file that holds the same name but another extension. Normal *.EXE files will get a *.COM companion. The system will also pick these up, and see whether an executable will create companions. If it does, the new companion must prove that it can recursively replicate in a clean sandbox.

WHAT CAN IT FIND?

The results coming out of the sandbox, can be one of three:

OK File is 'clean', nothing alarming found

MALWARE File modified binary executables or boot sequence on the virtual

storage device, but failed the recursive replication.

VIRUS File contains a virus, which replicated recursively.

It can't tell you what virus it is, but it can feed you some information about what it did. This is probably more interesting for the virus analyst, but also the administrator might see this as valuable.

As I've gone through zoo virus collections, there are quite a few files flagged as ok, but that scanners detect as viruses with name. Many of these files wouldn't work in the real world (stack problems, obvious bugs etc). In a strict sense, they shouldn't have been there. At least, they should have been flagged as intended. These files are not detected by this system, simply because they can't infect anything.

Since the program is already 'running' in the sandbox, you can detect malware that would be harmful to your system in a way that it:

- removes/modifies your already existing binary files
- attempts to reflash your BIOS (when it shouldn't)
- alters important registry keys
- simply attempts to trash your disk drive.

Malware is not easy to define. Things I look upon as malware might not be malware to another person. Executables that do permanent damage to file or system, I assume we can agree that it's not software we want to have on our systems.

As of now, the host program the virus is attached to, has impact on detection rates. If a memory resident virus re-starts the host, and the host crashes/never terminates, my OS can't tell what the resident portion would do. The virus will lay dormant in memory, and my 'drivers' will not get the chance of 'testing' whether a virus has gone resident. This problem has solutions. Once the OS sees that an executable has made the file system hooks, it can terminate the application in a matter of time if it doesn't terminate by itself.

Viruses are not always stable programs. They contain bugs, and some viruses are picky about what they infect, or when they want to do something.

- Viruses might execute the payload, and don't replicate
- Viruses may not want to go resident

- Viruses don't want to infect the goat files
- Viruses issue unknown APIs, interrupts, services or port requests.

After analysing a lot of viruses, you make a combination of everything that most real viruses will eat. *I don't have any virus specific code in the operating system* (e.g. a virus request that a byte holds a certain value), mainly to avoid conflicts with other viruses. My rule is: If it would work in the real world, it should work in the virtual world.

SELECTION OF GOATS

Goat files should be as small as possible, because it's time-consuming to load big files. However, many viruses only replicate to files of certain sizes, contain relocations or other criteria to make replication harder. To overcome this, you insert lots of different files on the virtual storage device. You don't copy them over each time, but you make logical links from the virtual storage device to the read-only copy you have in memory (definition file).

If viruses start modifying files, you make a read-write copy of that sector in memory, and redirect the logical link. In this way, you don't do too much work for every physical file you check. You can have a large number of files on the storage device without taking too many resources.

WHAT INFORMATION CAN BE EXPORTED FROM THE SANDBOX?

Since the system has logged the virus activity, it can give a nice description of how the virus works, what it infects, whether it's companion or not, etc. If dates are retrieved by the virus, the system can test the sample with various dates to see if the virus would carry any payload on a specific date. This is not implemented yet, but could be used as a preliminary virus analysis.

What information can easily be extracted?

- stealth: does it 'hide' information when you ask for it via the OS and compare to the physical data?
- memory resident: did it infect during execution of the sample or after? Did it install a file system hook via IFSMgr?
- infects boot sectors: were they modified during execution of the sample?
- which registry keys were altered to what?
- which operating system files or library files were modified during execution? Did kernel32.dll or wsock32.dll change?

What information could be gathered with some extra work:

- How did it infect files, could it be reversed to create cleaning scripts?
- Could it have any hidden payload, did it get system date/time and not do any thing?

BOOT SECTOR VIRUSES / MULTI-PARTITE VIRUSES

In general, these viruses are replicated in the sandbox as well. In the sandbox, there is also a drive unit 0, which corresponds to a regular floppy (A:). Viruses using either BIOS interrupt or talking directly to the floppy controller can infect this virtual floppy disk. If the floppy disk's boot sector has changed and no binary executables has changed, the CMOS will be set to boot from A:. If the hard disk system boot sector or master boot sector is now modified, it's safe to call the initial executable a virus.

Multi-partite viruses like Tequila, OneHalf etc., will modify the start up sequence of the computer, and wait for a boot to active the rest of the virus. My driver will continue to search for a resident virus, but no files will be infected. It will therefore issue a reboot. My BIOS can 'see' that the SBS/MBS is modified, and allows another go for the sandbox running the 'boot' portion of the virus. It will then infect my goat files, and these are again re-tested.

WHAT ABOUT THE VIRTUALLY INFECTED FILES?

We could easily export these files from the sandbox. Now, this is tempting but also very dangerous. These files are 'virtually' infected. Unless there are any bugs in my emulator, my kernels or my 'hardware' setup, this is what a regular infected goat would be like. There is no guarantee that it is so, and bugs can of course be around.

Introducing new variants is a bad thing, end of story. Whatever encryption we put on samples can be broken. As implemented today, we have no means whatsoever to take out samples from the sandbox at a user's site and ship them over to us for analysis. If we come up with a safe scheme of doing so, it might get implemented in the future.

PRELIMINARY VIRUS ANALYSIS

For the virus analyst having a new virus in front of him, this tool is worth gold. The analyst can, in a matter of minutes, determine what it does and what it doesn't. All the spies together with the ability to view the infected system will make it easier to understand the virus.

It's also a tool that can be used to replicate collections rather fast. It uses approx. six hours on a 'OneOf' collection consisting of 18761 files, having a replication rate of approx. 71%, and detecting approx. 5% as malware.

Total files 00018761

Time elapsed 05:58:14

Unknown malware 00000917 (04.88%)

Unknown viruses 00013409 (71.47%)

On average, it replicates approx. ten samples of every virus it's able to replicate. These goat files never leaves the AV-lab. They reside in a separate collection, and samples will not be given out

to anyone.

Our AV-lab version of NVC exports a log file from each infection, so you can make a huge database of all replications of a collection. It's easy then to search for specific events to find viruses using different techniques. Since this log file is on the interrupt service/API level, we will not display the log file since it might inspire people to start modifying viruses, or even write new ones. This system is designed to catch new and known viruses, not to help writing them.

QUALITY ASSURANCE

All objects (files, memory and boot sectors) can be modified inside the sandbox. If we have determined that we are dealing with a virus, we can also make sure we detect all instances of it using this system. All objects that are modified can be extracted to the real world (NOTE: it does NEVER happen on a user's system), and we can then test NVC to make sure it will find all files and/or boot sectors. Memory scanning and disinfection can also be tested/created. You can scan the 'sandbox' memory if you know your dealing with a memory resident virus. You can also attempt to remove it from the sandbox memory. When/if this is achieved, you make another effort to 'infect' even more clean files inside the sandbox. It they can't be infected, memory is cleaned.

As a test, I put our DOS-based behaviour blocker (NVC.SYS) active in the sandbox, and performed an on-demand zoo test of the behaviour blocker. Taking into account that this behaviour blocker was written in 1997, it still blocked approx. 60%. Now, blocked means that the virus was recognized in one way or another. Even so, the result is decent enough for an old retired project. We also had a *Windows*-based version of the behaviour blocker, but I haven't tested it so far inside the sandbox. It will be interesting to see the results.

SAMPLE SELECTION OF VIRUSES REPLICATED

I usually do tests on zoo collections. The most interesting are the viruses that so far don't have names, meaning they are new or unclassified. Out of 140 'unclassified' viruses, 22 were missed. Eight files were classified as malware, and the remaining 110 samples were detected as viruses, approx. 78% hit rate. And this is for sure – they were replicated recursively.

I've taken a cross-section of different virus approaches, and how the sandbox replication works for them:

CIH.1003 CIH is a resident Windows 9x/ME ring 0 virus. It will allocated memory through a VMM service, and install a file filtering hook. In this hook, it will follow all OpenFile calls, and infect using the RING0-API files with the PE header format.

Basically the virus does two APIs to go resident, and will then re-start the host it infected. Anyone performing a file I/O command will activate the virus. When the host program is terminated, control is passed back to KERNEL32!WinExec, which will return to the program requesting this application to run.

As a rule, my driver will now do a search on the virtual drive C:, for all executable files and perform an open. The find first/next, open & close are APIs exported from KERNEL32.DLL. These calls will eventually come as IFS packages towards the IFSManager. If a virus now is resident, it might try to infect the file I'm opening before I get access to it.

This is the danger many scanners have today when scanning infected systems without doing a proper memory scan – you risk that all executable files on your system get infected while your scanning for viruses. Resident viruses like CIH must be unlinked from the file system and removed from memory before scanning any files.

CIH monitors my open, and infects the file as I open it.

Since I now have a modified executable, I will re-test this one in a clean sandbox to verify that it will replicate in two generations. If the executable I was emulating wasn't a virus, my *Windows* would shut down, and my DOS would continue to verify the DOS stub of the PE file.

W95/Marburg. A: This is one of the first slightly polymorphic encrypted W95 viruses. It's direct action, meaning that it doesn't hook the file system. It will do all its harm when executed, and is not resident waiting for files to infected. Memory scan is not an issue with these viruses.

Marburg relies on many KERNEL32.DLL functions to work. It will locate these using Kernel32!GetProcAddress. It will delete some integrity-check files. It will then search your system for files to infect. If it finds one, it will memory map it using KERNEL32 functions. Basically it will infect the file in memory, and commit the changes to disk when done. It will use two APIs to write back bytes into the original host's process memory, and jump back to restart the host.

The process of detecting a virus like this doesn't make any changes to the basic logic. It's actually less work, since I don't need to 'find' the virus, using find first/next to make it replicate. When execution of the sample is completed, binary executables are already modified and are ready for re-testing for recursive replication

Dark_Paranoia.5554/5572: An old DOS virus using the CPU's ability to single step using INT 1. Each instruction the virus executes is decrypted before execution and encrypted again after execution. Thus you will never 'see' the virus unencrypted, but scanners do have techniques to detect these viruses exact. This type of virus doesn't pose a problem.

Level_3: This entire series of viruses causes no problem to replicate. Level_3 is really polymorphic, and goes resident in the sandbox. It infects files on close and execute in the sandbox.

Hare. 7786: Here is one of the viruses talking directly to the hard-disk controller to infect the MBS on your hard-disk. It also tries to delete your \WINDOWS\SYSTEM\IOSUBSYS\HSFLOP.PDR. It goes resident in the sandbox, and infects files using several writes.

THE NEXT STEPS ...

'Imagine a mail coming through a gateway system. It holds an attachment. The attachment is 'virtually executed' in the sandbox, and is stamped as a virus. The scanner engine will immedi-

ately start working out detection & disinfection. If it succeeds, the attachment will be cleaned, and the mail continues through. If it can't, it's deleted. The next time the same attachment comes through, the scanner has a signature to detect it, and already knows whether it can be disinfected or not. Everything necessary to do so is already located on the machine – nothing needs to be transmitted to any lab for analysis.'

This is what we're aiming at for the majority of viruses. It's not an easy task.

We have monitored the entire replication process and we have virtually infected files in memory. We can now let another module examine the layers of these files to find common data to draw signatures from. The signature will be tested first on all goats, and it will also go through a small 'false positive' test.

If the signature is accepted, it will be stored. The name of this 'unknown' virus will be a generic one. When a virus is detected with a signature that is made generically, the user/administrator must be alerted that detection was made with an automatic signature.

Since we have logged all events that have happened during infection of an executable, in most cases, it must be possible to create automatic disinfection. We can see where read/writes where done, into what logical areas the what file, etc. We also know how the original files looked like. On really complex viruses we will never manage to create automatic disinfection scripts. On the majority of viruses I think we have a chance of doing quite well.

When the scanner engine thinks it managed to make the disinfection algorithm, it can try it on the 'virtually infected files'. If these files don't turn out as the 'clean goats' they were prior to being infected, disinfection has failed, and we must try again even harder. In many cases, 100% disinfection is not possible and the system must be aware of this.

If the disinfection succeeds, the script used for disinfection is added to the signature and stored as well.

Worms pose threats that are bigger than any regular file infector. Using the Internet/open shares, worms can propagate to hundreds of users in minutes. How can we protect against unknown mass-mailers that don't use regular file replication? This scenario is still on the drawing board. If 'something' has modified e.g. WINSOCK32.DLL, my *Windows* driver will attempt to send a mail out to the Internet. Of course, there is no Internet, but as far as the computer/network believes, there is one. The mail is sent using WINSOCK/MAPI, and it should arrive to the my kernel in the same fashion it was sent, or else 'someone' has tampered with it. If it suddenly holds an attachment, it's alarming. If any mail is coming through without me sending one, it's also alarming.

Viruses/worms written in high level languages as Visual Basic cause problems since it's not trivial to write your own runtime DLL, which the executable expects. There are, however, shortcuts that might be possible, but I haven't pursued this further as of yet.

At a later stage I want to make my own Windows Scripting Host (WSH) which can run Visual Basic Scripts (VBS) inside the simulated *Windows*. I believe it should be possible, but it takes a lot of work to get it right. It would then be possible to examine the effect of running VBS. In theory, you could write your own WORD which can open a document, and see the effect on the system, but that will remain theory for a long, long while.

This system will never be finished. It will always pop up new challenges, but the important thing is that it's something to build on. It's not meant as a replacement of today's AV solutions, but as an add-on. You can extend the architecture, building new blocks and modules to detect new classes of viruses on new platforms.

All it takes is work...