

VIRUS ANALYSIS

The Marburg Situation

Péter Ször
Data Fellows

While the number of 32-bit *Windows* viruses is not rising quickly, it is alarming that three have reached the WildList recently. We first saw Win95/Anxiety (see *VB*, January 1998, p.7), then different variants of Win95/CIH (*VB*, August 1998, p.8) and now Win95/Marburg is on the list. The latter is the first in the wild, polymorphic virus to infect only Portable Executable applications. Few *Windows* viruses have been this successful – most have been full of bugs and thus unlikely to become widespread.

Where Will Viruses Want to Go Tomorrow?

The appearance of Marburg shows that *Windows 9x* viruses do have the potential to spread as quickly as, or even faster than, boot viruses. [Although this analysis appears after that of CIH, Marburg predates CIH by several months. Ed.] DOS is not the PC-dominating platform it was, having been replaced with different implementations of *Windows*. When the dominant platform on popular computers changes, some virus types will die out but this will not mean an end to the virus problem. When a new platform begins to dominate, we see thousands of viruses for it in just a few years. This is likely to continue on Win32 platforms – first on *Windows 9x*, later on *Windows NT*.

A year ago we did not see more than one *Windows 95* virus per month. By the middle of 1998 this had changed to an average of about one per week. Nowadays, there is more than one variant per week. During 1999 we may see a new *Windows 9x* or *NT* virus almost every day. So far this is what we have seen with other virus types and nothing suggests it should be different with 32-bit *Windows* viruses. This is not helped by some virus writers realizing just how easy it is to develop viruses in a high level language such as C or even Delphi. [...and VBA, of course! Ed.]

There are now several viruses that are around 100 KB to almost half a megabyte long. This is not the case with Marburg, however. Marburg is written in assembler and was probably the first polymorphic *Windows 9x* virus. Its author also wrote Win95/HPS (*VB*, June 1998, p.13). The Marburg polymorphic engine is very similar to that of HPS, but looks like an earlier development. While HPS hooks system functions, Marburg is a direct action infector. The balance of probabilities suggests Marburg must have been written and released long before HPS for it to be in the wild. HPS has not been seen in the field so far.

Marburg's current claim to fame is that files infected with it were found on several magazine cover CDs. First, it was included accidentally on the cover CD of the July 1998

edition of UK-based *PC Gamer* magazine. A utility program that was automatically executed if you chose to watch any of the preview videos from the CD was infected. Localized versions of *PC Gamer* exist, in addition to the UK edition. The Swedish and Slovenian editions also carried infected files.

Then, in August, Marburg was included on the master CD of the popular *MGM/EA* game *WarGames*. It also had widespread circulation on the cover CD of Australian *PC PowerPlay* magazine in August 1998. [A further incident is listed in the October issue Editorial. Ed.]

This seems to be the most 'successful' distribution of a virus, at least in such a short period of time. Unlike many viruses, Win95/Marburg has few bugs but, fortunately, one of them is fatal enough (if very small) to prevent the virus replicating under *Windows NT*. The virus contains the text '[Marburg ViRuS BioCoded by GriYo/29A]', hence the name Win95/Marburg.A. The .B variant is also unable to replicate under *Windows NT*.

Why did Marburg get the opportunity to spread so widely and end up on so many commercial CD-ROMs? As it happens, most scanning engines had to be changed in order to detect this virus reliably. Such major changes always require a longer development time than simple database updates. There are still only a few products which can detect Win95/Marburg.A in all circumstances.

The virus utilizes a slow polymorphic replication mechanism. Further, the infection method differs slightly in some files. This small difference may not have been apparent to some virus analysts at first glance. A few missed samples on each PC can be enough to keep the virus circulating over and over. Deliberately targeting screen saver (SCR) files may also have assisted distribution.

Executing an Infected Application

Win95/Marburg.A is a PE infector. When an infected 32-bit application is executed, the virus code takes control. When the host program does not have relocation for the first five bytes at its entry point, the virus places a jump instruction there and does not modify the entry point field in the PE header. Otherwise, if it is really necessary because there is a relocation for the first instruction, it modifies the entry point in the PE header and the code at the original entry point remains the same.

Then comes the trickiest case. When there are no relocations for the first 255 bytes from the entry point, the virus not only places a jump instruction in the code at the entry point of the host, but builds a random garbage code block first and puts the jump to the virus' polymorphic decryptor at the end of it. The size of the junk block

together with the jump will be less than 255 bytes. The jump instruction or the entry point field of the PE header points to the very end of the real virus body which is always attached to the last section of the host program. Then the polymorphic decryptor decrypts the virus body which precedes it.

The size of the virus body (without the decryption code) is a constant 5793 bytes but infected files will grow by around 7900 bytes. This is because the size of the polymorphic decryptor and the constant virus body is 7841 bytes and the virus pads itself out to make the infected file size exactly divisible by 101. Several viruses written by members of the 29A group use the same self-recognition technique to prevent multiple infections.

Marburg uses several techniques with similar functionality to that of Win32/Cabanas (see *VB*, November 1997, p.10). Marburg attempts to save pointers to the import addresses of the `GetModuleHandleA` and `GetProcAddress` APIs during the infection process. Once the virus body is decrypted and control passes to it, if these API addresses are available in the host program's import table, Marburg can work easily.

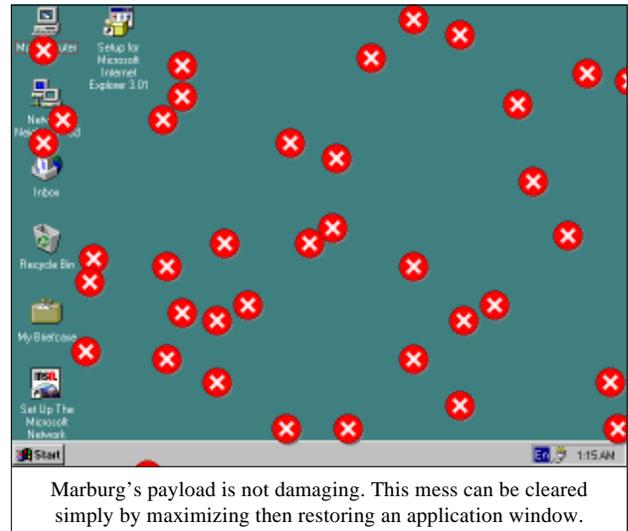
When the address of `GetModuleHandleA` is not available, as in Cabanas, Marburg tries to use the `ForwarderChain` field of the import table. At that moment the virus knows the base address of the loaded `KERNEL32.DLL` in the virtual address space of the process. When the host program does not have imports for the `GetProcAddress` API, the virus simply searches the export table of `KERNEL32.DLL` and picks up the address from there.

After that, the virus is in a position to obtain all the API addresses it needs from `KERNEL32.DLL`. Altogether there are nineteen APIs of interest to the virus (including `CreateFileA`, `CreateFileMappingA` and `GetSystemTime`) and it gets all of them by calling the `GetProcAddress` API in a loop. If an error should occur, the virus executes the host program. Before executing the host, Marburg checks whether fixes are needed at its entry point. If so, the virus replaces the code at the host's entry point with the original code, then passes control there.

If no errors occur, Marburg allocates memory and copies itself there, passing control to that copy of itself. This mechanism is needed because of the virus' polymorphic engine. During infection, Marburg saved the current date and hour in its body. At this point of execution it checks whether the infected program is being run three months after its infection and during the same hour. Whenever these conditions are met, Marburg calls its payload.

Spots Before Your Eyes?

The payload routine needs the addresses of three APIs from `USER32.DLL`. The virus first ensures this library is loaded, then uses `GetProcAddress` three times in succession to call the required APIs.



The first API is `LoadIcon`. Marburg loads the standard `IDI_HAND` (0x7F01) icon resource which *Windows* uses in the case of serious error messages – a white cross on a red circle in the case of *Windows 9x*. Then it gets a handle to the desktop with the `GetDC` API. Finally, it draws up to 255 icons (depending on the screen resolution) at random positions on the desktop.

Windows 9x will gradually redraw the desktop area as window sizes change, causing Marburg's icons to disappear. However, the same infected program will display some icons again in the same hour, as will other infected applications, should the payload trigger conditions be met.

Infection

Marburg is a direct action virus. The virus tries to infect one file with an `EXE` or `SCR` extension in each of the current, *Windows* and *Windows System* directories. Marburg is a retro virus, deleting known checksum files of different anti-virus products such as 'ANTI-VIR.DAT', 'CHKLIST.MS', 'AVP.CRC' and 'IVB.NTZ' in every directory it attempts to infect. The virus avoids infecting many anti-virus programs. It also avoids infecting any program whose name contains the letter 'V' or strings 'PAND', 'F-PR' or 'SCAN'.

Before infecting a file, the virus changes the file attribute to normal. Thus, it can easily infect read-only files. The virus uses file mapping functionality which makes the infection process much shorter than it would be otherwise. Marburg always places itself into the last section of the host. However, it does not infect the file if the last section has shared characteristics. It sets the last section characteristic to include the writeable attribute.

The infection procedure is protected with Structured Exception Handling (SEH), thus the virus will execute the host program if a GP fault should occur in its own code. Viruses using this technique can be very stable (and more successful). During infection, the virus checks if the host

program is 386-compatible and only infects it if that is the case. It tries to save references to the `GetModuleHandleA` and `GetProcAddress` import addresses, as this makes the initialization less complicated later on. Then it checks the relocations and according to those uses different entry point strategies as described above.

Before infecting a file, the virus calls its polymorphic decryptor generator. This engine implements slow polymorphism, thus several infected files on any infected PC will have the same polymorphic decryptor. Further, the number of different combinations is limited compared to what the engine could generate.

In any case, the virus' polymorphic engine is powerful, using several different encryption methods and keys. The size of the polymorphic generator is 1872 bytes and, as already noted, is similar to that of Win95/HPS, but somewhat limited in comparison. When the polymorphic decryptor is generated, the virus encrypts its main body and writes everything into the host. At the end of each infection, the virus changes the host's file attributes back to their original state. Finally it executes the host program.

Conclusion

The wide distribution of Win95/Marburg shows that a well written, complex, polymorphic virus can be successful. Such complex viruses have not appeared in the wild before, but that situation may be about to change dramatically.

Anti-virus researchers have to invest lot of energy and time into a complete analysis of such new viruses in order to design correct detection and disinfection methods for them. DOS viruses with direct action infection mechanisms do not usually spread too far. This situation seems to be different in case of multi-tasking environments and that makes the job of virus writers even easier.

Marburg

Aliases:	None known.
Type:	Windows 9x direct action, PE infector targeting EXE and SCR files.
Self-recognition in Files:	Files whose size can be divided by 101 without remainder are assumed to be already infected.
Hex Pattern in PE files:	Not possible.
Payload:	Displays the default error icon at random positions on the desktop during the hour of initial infection, three months after that date.
Removal:	Recover infected files from backup or replace with original.

FEATURE

The Biggie

Peter Morley
Network Associates

Back in 1992, when virus authoring packages first appeared, people who worked in virus labs became aware of a potential nightmare. What if someone used such a package to produce many thousands of new viruses? How would we cope? Anti-virus people were wary of discussing it in print.

It took a while to happen, but during the last week of September this year, Dmitry Gryaznov told me that it had. From the virus writer's point of view, it had got to be a case of checking the stable door well after the horse had bolted over the hill!

On Friday 2 October, we received the largest collection of viruses which has ever come our way. It consisted of almost 15,000 viruses that we had never seen before. It had been generated in the field and sent to several anti-virus developers and testers.

We completed our processing of this collection within a week. This article explains how we did it, and some of the thoughts which went into deciding the process, given that in our case, the result had to be not only detection of the 15,000 files, but also detection and repair of anything which is produced from them.

First Thoughts

How about:

- Ignore it, and hope it will go away.
Well, I couldn't get away with this, even if some of our smaller competitors could.
- Issue a Press Release, claiming we already detect most of them, and do no work at all. Just continue processing macro viruses. Anyway, they are not in the wild so they will never be a problem, will they?
I think you already know my opinion of anti-virus people who hide behind an imaginary wild list!
- How about the classic approach? Replicate each one which will replicate, and process it normally. We can do all the replication without manual intervention. But what do we do then? We will have a lot of files!
We do not have an IBM automatic processing system, and even if we did, might this possibly clog it up?

Second Thoughts

Hang on a minute. We would have to do the unavoidable. Examine the problem, and *think*. The first three stages are standard, starting with the elimination of all duplicate files.