

# VIRUS ANALYSIS

## Tasting Donut

Péter Ször

Symantec Security Response, USA

In early June 2001, several anti-virus companies received a copy of a new virus from its author. It was quick to be described by certain AV companies as ‘the first known virus implemented in the Microsoft C# and Microsoft Intermediate Language (MSIL)’. However, this claim is simply incorrect.

Although the virus has a few bytes of MSIL code, the actual virus code is 32-bit assembly. The virus body does contain some MSIL code, but this is very short (a dozen or so MSIL instructions in the entry-point method) and is not involved in any way with the replication functionality of the virus. Furthermore, there is a minor bug which prevents the trigger function from working.

Regardless of this, the virus is interesting enough to take a closer look at, since it really is the first virus known to attack .NET applications.

The author of this virus, who goes by the name ‘Benny’, wanted it to be named ‘dotNET’. Obviously such a name would not be acceptable, so I decided to call the virus ‘Win32/Donut’.

### .NET Framework

Two excellent papers addressed the forthcoming .NET virus problem at the *Virus Bulletin 2001* conference (Philip Hannay and Richard Wang’s ‘MSIL for the .NET Framework’ and Eric Chien’s ‘Effects of Microsoft .NET on Malicious Threats’). However, it appears that a few things have changed in the file format since then and several things became obsolete straight after Beta 1.

The .NET architecture uses regular Portable Executable file format. The executable supports a platform-independent code. The actual code of the application is MSIL, which is compiled to native code on the fly by the JIT (just in time) compiler.

The Beta 1 file format is no longer supported on Beta 2. The .NET PE files use a six-byte platform-dependent code, a jump to a statically imported DLL function `_CorExeMain()` in `mscorlib.dll`. This routine will initiate the Common Language Runtime (CLR) and thus the MSIL portion of the EXE applications. (DLLs will use a `_CorDllMain()` function.) The CLR is able to locate the MSIL code as well as the metadata via the CLR header defined as `IMAGE_COR20_HEADER` in the .NET SDK and available via the fourteenth entry (+0x70) of the data directory.

These are placed here for operating systems that are runtime-unaware and that are not marked for CLR-only execution. Evidently, Microsoft is willing to extend the system loader of its upcoming operating systems to support the file format without the need of a startup call nor relocation section placed in the image.

Thus, according to the .NET SDK, these portions of the PE files can be safely ignored by future operating systems. However, for older systems this structure will provide backward compatibility.

### Gaining Control

Donut gains control immediately upon executing an infected EXE file. The virus uses the simplest possible infection technique to infect .NET images. In fact, Donut turns .NET executables into regular-looking PE files. This is because the virus nullifies the data directory entry of the CLR header when it infects a .NET application.

The six-byte jump to the `_CorExeMain()` import is replaced by Donut with a jump to the virus entry point. The entry point in the header remains unchanged. In the past some of us in the AV industry considered this method to be an entry point obscuring technique – we call it ‘obfuscated tricky jump’ these days. Evidently this technique will fool some of the heuristics scanners.

The actual jump is a 0xE9 opcode followed by a DWORD offset to the start of virus body in the first physical byte of the relocation section.

### Initialization

First an exception trap is created to protect the code from unwanted GP faults. After this Donut attempts to identify the location of a loaded `KERNEL32.DLL` using a pretty much standard approach found in 32-bit Windows viruses. In fact, the function handles a couple of operating systems correctly, including *Windows NT/2000/XP* and *Windows 9x*.

Donut uses CRCs of the APIs it wants to call. This saves a few bytes and makes analysis little more complicated. Altogether there are 24 API CRCs in the virus body, starting with `GetVersion()`. The list ends with `DeleteFileA()`. However, some of the APIs, such as `GetSystemDirectoryA()` and `SetFileAttributesA()`, will not be called from the list. It is possible that the virus writer copied this section from another virus and these were remnants of that code.

Next, the virus checks the version of the OS. If the major OS version is not at least 5 the virus will not infect. Instead it will attempt to execute its trigger function and eventually the host. However, if the OS version is 5 or above the virus

will attempt to infect all files with a '.exe' extension in the current directory and 20 directories above it, using a direct action method.

Since the host application has a static import to the mscoree.dll the virus will not be able to load if the .NET framework is not installed on the machine.

### Infecting a File

First the infection routine checks the file size. If the file is too large (bigger than 4 GB) or too small (smaller than 2048 bytes), the file is ignored. Otherwise the file is opened and mapped.

Next the file is checked for MZ mark and the PE mark using a tricky comparison. If the PE file is not a 386 image the file is ignored. Similarly DLLs, system and native applications are also ignored.

Now the relocation directory entry is checked. If there is no relocation section the file will be ignored. Otherwise the relocation entry in the data directory will be nullified. Thus infected files will no longer be considered for infection. Donut also ignores files that have a single section.

Next the virus looks for the CLR header entry in the data directory. If the RVA of the CLR header is not 0x2008 and its size is not 0x48 bytes, the virus will ignore the file. Similarly, the file is ignored if the image base is not 0x400000. Files that do not have an import address directory at 0x2000 are also ignored.

All of this is in order to search for applications compiled with the C# compiler. Thus Donut will not infect all .NET applications, but only a subset of them with a very specific format.

The Beta 2 .NET files have a 'BSJB' CLR header signature. The virus searches for this and therefore skips the Beta 1 files. The CLR entry in the data directory is also nullified. The entry point is replaced with a jump to the last section and the relocation section is overwritten with the virus code.

The virus saves the indirect pointer of the jump at the entry point, assuming a two-byte (0xFF25 opcode) jump right at the entry point. Again, this is an assumption for C# compiled executables. Since the virus does not check the stub code, in the case of a .NET file with an unusual entry point code the virus would fail to start the host correctly.

### File Size

Donut saves 862 bytes starting at the CLR header and overwrites this with its own MSIL trigger function. The actual size of the 32-bit assembly portion of the virus code is 2180 bytes but the virus carries its trigger metadata (862) bytes, plus it will save the same amount from the original host. Thus the total size of the virus with the saved host data is 3904 bytes.

Since the virus will overwrite the relocations, in cases where the relocation section is sufficiently long the file might not increase in size after infection. This is very unlikely though.

In the case of smaller files, and with most C# compiled executables, the file size will increase by 4 KB after infection. This is because the virus uses a 4 KB alignment to make the image acceptable for the CLR. In addition the size difference is corrected in the virtual size field of the last section header.

Next the last section is marked writeable. At this point the infection is almost complete but Donut calculates a proper checksum for the infected file by loading imagehlp.dll and using CheckSumMappedFile() API.

Finally the file is closed and the exception handling is removed. Normally this procedure should continue until each file is infected in the current directory and 20 directories above it.

### Executing the Host and Trigger

Eventually the host would need to be executed. It seems Benny had the bright idea of not only executing the host application but executing the MSIL trigger routine too. Thus the procedure that originally functions as a routine to execute the host was modified as well as the infection routine.

However, Benny forgot about one thing which causes the functionality of the virus to change quite considerably.

Benny wanted to execute the MSIL routine so he copied it to the right place during infection. Obviously infected files need to start with the virus. Therefore the entry point code is replaced with a jump to the virus body.

On attempting to execute the trigger routine Donut makes a copy of the infected file. For example, on execution of 'runme.exe' the virus will create a copy of the file as 'runme.exe' using CopyFile() API.

This image is mapped and its CLR header is fixed by assigning 0x2008 and 0x0048 values to the data directory entry. The virus uses the APIs CreateProcess() and WaitForSingleObject() to execute that image and to wait for its termination.

However, the fix of the entry point code is missing. Since the virus writer failed to think about this problem, Donut will execute itself again. Thus a new file will be created, say 'runme .exe', followed by another file, 'runme .exe', 'runme .exe' and so on.

Only after this step will the entry point code be fixed and executed again to run the original application. Thus the original host could execute many times and the MSIL trigger routine would never gain control in the CLR ... Oh well.

Basically the virus is in an endless recursion. Eventually, however, the file name becomes too long and cannot be created.

This will stop the cascading effect and results in the execution of the host application sometimes more than 200 times. At least the host is executed with proper command line parameters. But is this not a little too much?

Eventually all the temporary files will be deleted by their executor so the mess is cleaned up in the current directory appropriately.

### Getting the Message

The trigger would have displayed a message box with a ten per cent chance. So, should the trigger function work, how would this message look?

A wide variety of interesting guesses from the AV industry can be found posted all over the Web. This is either because the virus was not replicated or the virus analysts might have simply considered themselves unlucky. I can only guess that a short VBS script did the trick for those nice write-ups.

The actual virus code would result in the following message box being displayed:



### Conclusion

We probably need to wait a little longer until 'the first known virus implemented in the Microsoft C# and MSIL' is created. This could take a little more effort on the part of virus writers since it is not a simple matter at all.

In the future I would expect to see worms written with backdoor features that utilize the .NET networking features. Viruses that infect other .NET files might not have the opportunity to become a major problem.

Name: W32.Donut	
Type:	Win32 direct action virus.
Removal:	Replace infected files from backup.
Trigger:	Attempts to execute an MSIL written application to display a message box with 10% chance, but fails to do so because of bugs.