

## VBA Emulation: A Viable Method of Macro Virus Detection? Part Two

*Gabor Szappanos* 2002-05-02

### VBA Emulation - A Viable Method of Macro Virus Detection? Part Two

by *Gabor Szappanos*

last updated May 2, 2002

---

This is the second of two articles discussing emulation as a viable method of virus detection. In the [first article](#) we briefly examined how emulation worked and began a discussion of some of the problems of emulation, particularly with macro source and macro execution. In this article, we will discuss code execution flow, underlying operating system problems, and incompatibility issues with incompatibility in different versions of Office, as well as VBA emulator environment.

## Code Execution Flow

Unfortunately, there are already cases that will not fit into the simple execution order discussed in the first article. Some viruses use more exotic infection schemes. The following section will look at some examples of this.

### Office XP Macro Viruses

We can expect the new Office XP aware macro viruses to use a different two-phase model. In short, the default settings of the macro virus protection in Office XP prevent any access to the VBA object model. This effectively stops the macro viruses from injecting their code into the global template or into other documents. This access block can be disabled (giving way to viruses) by changing a single registry value. Word reads this value once upon start-up. For this reason, the Office XP macro viruses, like WMXP.Listi switch off the macro virus protection on the first run only, then, on the next activation, they can freely infect the global template and other documents.

The emulator should run the sample at least two times, registering the changes in the system registry.

### Run-time Error Handling

Once we have solved all the problems regarding the entry point to the macrocode, it would seem that it is a simple job to follow the execution flow: being an interpreted language, it is clear what the next instruction will be:

- If the current instruction is a procedure call, then the next instruction will be the first instruction of the called procedure;
- If the current instruction is an execution control instruction (Goto, Gosub, ...), the execution should skip to specified label or line number (VBA is still BASIC: it supports line numbering that act as labels for the line);

- If it is a loop closing instruction, and the loop abort instruction is not met, the execution goes to the beginning of the (For, While, Do) loop.

Several macro viruses display dialog boxes during infection and, depending on the user input, or depending on the command button the user presses, the execution forks. Otherwise, the execution goes to the next line of code. Usually. Except when a runtime error occurs. Then the code execution could go several ways, depending on the error handling conditions.

There are several ways a macrocode can handle the errors. This first and most simple is the in-line error handling, achieved by the "On Error Resume Next" statement. This simply drops the error condition and continues the execution with the next command line as briefly explained above.

A more sophisticated method is to set up an error trap with the "On Error Goto" instruction. This method, being somewhat more complex, is not widely used in macro viruses (though interestingly enough the WM. Concept used it). The "On Error Goto eh1" instruction specifies the label (or line number) where the execution should follow when a trappable error has happened. Then having this error occur, the execution skips to label "eh1", where first the error condition is cleared, after which some action is taken, finally the normal execution path is resumed.

This process is illustrated with the following code snippet.

```
Sub test ()
On Error GoTo eh1
test1
10 MsgBox "No error"
GoTo e_end
eh1:
20 Err = 0
MsgBox "Error!"
Resume e_end
e_end:
End Sub
```

```
Sub test1()
On Error Resume Next
Err.Raise 220
30 MsgBox "Still here"
End Sub
```

So whenever an error occurs, the execution either:

- aborts if there is no error handling initialized;
- skips to the next line, if the in-line error handling is initialized; or,
- skips to the specified error trap.

Unfortunately, the situation is somewhat more complex than this. An error handler's context is extended to the procedures that are called from the procedure that initialized the error handling - unless the called procedure overrides it. In the previous example: if the "On Error Resume Next" instruction is present in the procedure test1, then, after the simulated error occurs, the execution follows to line 30. Finally, after returning from the call it will go to line 10. However, if the "On Error Resume Next" is not present in the "test1" procedure, then after raising the error, the execution immediately skips to line 20. Therefore, the execution flow could go different ways depending on the currently defined error handling: it may even abort the current procedure and return to its caller or even to the caller of the caller, if there were no error handlers defined there. It could even fall back higher in the call hierarchy - until the first valid error handler is found.

Therefore, the emulator should also emulate the runtime errors and the chained list of error handling procedures.

## Underlying Operating System Problems

Viruses read and write registry and their action depends on the values read back. A simple registry also has to be emulated with the most basic settings. For initial reading, the original registry would be appropriate - if the emulator only ran under Windows platforms. However, virus scanner exists on non-Windows platforms as well; therefore, the emulator has to use a shadow-registry containing the basic settings of a default Windows+Office installation. Not to mention the fact that if the emulator used the existing registry, the virus scanner behavior would depend on the PC on which it is actually installed, which is clearly nonsense.

Viruses also make use of the Windows API functions (for example to manipulate the registry). The full-scale emulator should also emulate a minimal (and far from being complete or sufficient) set of Windows API functions, such as:

```
Private Declare Function ShowCursor Lib "user32" (ByVal bShow As Long) As Long
Private Declare Function FindWindow Lib "user32" Alias "FindWindowA" (ByVal lpClassName As String, ByVal lpWindowName As String) As Long
Private Declare Function ShowWindow Lib "user32" (ByVal hwnd As Long, ByVal nCmdShow As Long) As Long
Private Declare Function GetDriveType Lib "kernel32" Alias "GetDriveTypeA" (ByVal nDrive As String) As Long
Private Declare Function GetLogicalDriveStrings Lib "kernel32" Alias "GetLogicalDriveStringsA" (ByVal nBufferLength As Long, ByVal lpBuffer As String) As Long
Private Declare Function GetVolumeInformation Lib "KERNEL32" Alias "GetVolumeInformationA" (ByVal lpRootPathName As String, ByVal lpVolumeNameBuffer As Long, ByVal nVolumeNameSize As Long, lpVolumeSerialNumber As Long, lpMaximumComponentLength As Long, lpFileSystemFlags As Long, ByVal lpFileSystemNameBuffer As Long, ByVal nFileSystemNameSize As Long) As Long
Private Declare Function RegCloseKey Lib "ADVAPI32.DLL" (ByVal hKey As Long) As Long
Private Declare Function RegOpenKeyEx Lib "ADVAPI32.DLL" Alias
```

```

"RegOpenKeyExA" (ByVal hKey As Long, ByVal lpSubKey As String, ByVal
ulOptions As Long, ByVal samDesired As Long, phkResult As Long) As
Long Private Declare Function RegQueryValueEx Lib "ADVAPI32.DLL" Alias
"RegQueryValueExA" (ByVal hKey As Long, ByVal lpValueName As String, ByVal
lpReserved As Long, lpType As Long, lpData As Any, lpcbData As Long) As Long
Private Declare Function RegSetValueEx Lib "ADVAPI32.DLL" Alias
"RegSetValueExA" (ByVal hKey As Long, ByVal lpValueName As String, ByVal
Reserved As Long, ByVal dwType As Long, lpData As Any, ByVal cbData As Long) As
Long
Private Dclare Function GetTempPath Lib "kernel32" Alias "GetTempPathA" (ByVal
nBufferLength As Long, ByVal lpBuffer As String) As Long
Private Declare Function CopyFile Lib "kernel32" Alias "CopyFileA" (ByVal
lpExistingFileName As String, ByVal lpNewFileName As String, ByVal bFailIfExists
As Long) As Long
Private Declare Function GetTempFileName Lib "kernel32.dll" Alias
"GetTempFileNameA" (ByVal lpszPath As String, ByVal lpPrefixString As String,
ByVal wUnique As Long, ByVal lpTempFileName As String) As Long

```

So, which OS platform should we choose: Windows or Mac? There are some differences between these platforms that result in different behavior of the executed virus code in real-world applications. In addition, the viruses that use Windows API functions are restricted to one operating system. Therefore, the emulator should have plug-ins for at least a Windows 98 and a Windows 2000 operating system, as well as an additional Mac OS.

The default operating system should be an English-language Windows plug-in, but if during the processing any signs shows that the virus requires a different OS (different API calls, UNICODE characters, separators), the emulator should switch to the required operating system plug-in and restart the emulation.

## Office Version Incompatibility Issues

The emulator has to make a decision, which Office version to emulate: Office 97, Office 97 SR1, Office 2000 or Office XP? It is well known to the audience that there are differences between these versions. There are two of them that are highly relevant for macro virus behavior: the protection introduced in Office 97 SR1 and the automation block in Office XP.

The VBE object model provides several methods for manipulating VBA code. Office 97 SR-1 disabled only one of these methods, the use of the OrganizerCopy and the WORDBASIC.MacroCopy (which was the unconverted version of WordBASIC's MacroCopy) method to copy macrocode from the normal template into the active document.

The basic and default plug in should be the Office 97 pre-SP1 version, as this provides the most virus-friendly environment. If any sign shows that the virus requires a higher Office version, a different plug in should be loaded.

Several viruses check the Office version during execution, and take actions depending on this version. The

emulator should recognize these attempts, first return the Office version of the default plugin, then if the suspected code did not spread, it will have to load the next plug in emulate again and on return the appropriate version number.

## VBA Emulator Environment

### The Emulated Environment

The emulated environment should provide the object models for the most common applications and objects (with their methods and properties) used by macro viruses. As the number of these is huge, one has to make reasonable compromises. Only a limited, minimal set of objects is emulated, and a limited set of properties and methods are supported. The rest are simply ignored. The bad news is that the parser still has to recognize most of them; otherwise it would treat them as variables instead of built-in objects. For example, it makes no sense to emulate the Fonts object that is present in the Office applications, as it is not apparent from the virus's behavior which formatting it would apply. The VBProject object should be entirely emulated as present and future viruses make use of all of its methods and properties.

Not only should the Office applications be integrated into the environment, theoretically, any ActiveX server application is should be there, and at least the most important ones used in viruses should be added. Most importantly, the Windows Scripting Host object (namely the Shell object and the FileSystemObject) belongs here, not only because we want to process VBScript worms, but also because even Office macro viruses use these objects for example for registry manipulation.

A minimal and incomplete list of necessary applications and objects (the collection objects are omitted as but these should be there accompanying the objects that belong to collections) is listed below:

<b>VBIDE</b>	<i>VBProject</i>
<b>Microsoft Word</b>	<i>Application, Document, Addin, Template</i>
<b>Microsoft Excel</b>	<i>Application, Workbook, Worksheet, Addin</i>
<b>Microsoft Outlook</b>	<i>AddressEntry, Attachment, NameSpace, MailItem, Recipients</i>
<b>Microsoft PowerPoint</b>	<i>Addin, Presentation, Shape, ShapeRange, Slide</i>
<b>Microsoft Office</b>	<i>CommandBar, FileSearch, COMAddins, FoundFiles</i>
<b>WSH</b>	<i>Drive, Folder, File, FileSystemObject</i>

## Macro Viruses and the Global Template

By definition, we should consider macro viruses to be those macro programs that, after running, create code in the new macro modules. It would be tempting to stop when macrocode appears in the emulated normal template; unfortunately, there are a couple of self-installing utility macros (some of which are virus protection tools) that install by copying their macros into the global template. As we want to avoid false positives, a

stricter rule and procedure is needed. We should not forget the "non-resident" direct action macro viruses that do not infect the global template, only the currently open documents.

Thus, the emulator environment would consist of the global template, a couple of open clean goat documents, and the inspected document. The described event scheme is emulated twice in a row, first with the inspected document being open, then with it being closed. In the second run, the subject of the event would be one of the goat documents. After the second run, the integrity check is performed. If macrocode appears in any of the goat documents, it is relatively safe to assume that we have a virus. If macrocode appears in any of the emulated additional application environments (like in the case of cross-application macro viruses), the affected environment should also be "started up" and its default event chain is executed. As the virus could hop back and forth between the applications, the engine should not allow re-activation of an already emulated application.

During the second run, we assume an application restart. This means that the emulator should reload the startup templates (there are none by the default configuration) - a large number of Excel viruses operate by creating a startup template. During the event chain execution, we should take care in the normal storage precedence order of Office: Active document -> Loaded startup templates (in alphabetic order) -> Global template.

## Conclusions

After all these considerations it should be obvious to the audience that the emulator environment is highly complex. For the time being, it requires too much time for subsequent runs to be viable solution now. However, as malicious code becomes more sophisticated, anti-virus defences must evolve accordingly. Although emulation is still in its infancy, developers will almost certainly iron out the many kinks that preclude widespread usage. eventually it has to be considered for implementation.

### Relevant Links

[VBA Emulation: A Viable Method of Macro Virus Detection? Part One](#)

*Gabor Szappanos, SecurityFocus*

[Privacy Statement](#)

Copyright 2006, SecurityFocus