FEATURE

‘In the Beginning was the Word...’

Andrew Krukov

The twentieth century has been one of innovation and new technology, seeing the popularization of the so-called ‘thinking machines’ we call computers. A side-effect of this development is the computer virus: today, approaching the end of 1996, viruses have been infiltrating machines for over ten years, replicating, crashing systems, and corrupting data.

Physics teaches us that every action has an equal and opposite reaction – so, following close on the heels of viruses, along came the anti-virus industry. Many hundreds of people the world over work for dozens of research companies and hundreds of sales/support sites which, with varying degrees of success, protect users against computer bugs.

Millions of dollars are lost to virus action, and millions are spent to recover data, or to buy anti-virus and other security soft- and hardware. Hackers write viruses, other hackers create new anti-virus programs, publishers print them, distributors distribute, end-users buy. And life goes on…

The End of the Beginning

What would happen if the world woke up one sunny morning, and viruses had disappeared overnight? Perhaps not much of significance. Users would be happy, the anti-virus industry would put its expertise into other fields, and Grandma would tell small children old myths about the nasty viruses which used to run rampant through the computers of the world.

Is it possible to kill viruses off forever? To do this, everyone would have to use operating systems that do not support viruses. Granted, viruses may be written for any popular OS; but to write viruses and spread them internationally, two things are necessary:

- a well-documented OS, which makes the writing easy
- many people exchanging executables for that OS

Only one OS meets both requirements: the very popular (and remarkably fully-documented) DOS. DOS viruses are the only ones, in the last ten years, to have created problems daily for users in every corner of the planet. *Windows* viruses were discovered in the wild only in 1996, and the Tentacle variants are the only ones to make any impact so far. No *Windows 95* or *OS/2* viruses are in the wild.

Moreover, compared with the circa ten thousand DOS viruses, the number of viruses for other operating systems is paltry: 100–200 *Mac* viruses, fewer than twenty *Windows* viruses, three *Windows 95* viruses (all are variants of Boza), and a handful for *OS/2*. So, there are over 100 times as many DOS viruses in existence as the total of all other viruses.

Therefore, it seems that to break the circle of virus writing, users must stop using DOS and turn to one of the plethora of new operating systems. Viruses will then die, as will the anti-virus industry.

Not so.

A Totally New Concept

WinWord.Concept overturned these beliefs. This infector was the first in the new breed of *Word* macro viruses; viruses for which the old rules do not apply. They are application-specific, multi-OS viruses; spreading only within *Word* documents, but under all OSs for which a version of *Word* is available.

They are at the same time simple and complex: simple, because they are written in a variant of Basic, so it is not necessary to look at long listings of assembler instructions to analyse them; complex, because locating the infected macro in the document, detecting the virus and disinfecting the document is a complex task. To make matters worse, *Word* macro viruses spread like wildfire – after all, *Word* documents are a standard method of data exchange.

So anti-virus researchers began to direct their considerable resources and intellect against the new ‘visitors’. To detect and disinfect these viruses, it is necessary to parse the *MS Word* format, then go through data structures, calculate pointers, follow these pointers, and examine a considerable amount of data – and all this simply to *find* the macros in a given document!

The binary format of a *Word* document is more complex than that of a conventional executable. A *Word* document looks like an entire filing system, with its own FATs, directories, blocks of data, etc. Researchers have spent a great deal of time, and used many different techniques, to reverse-engineer this format; to understand this most undocumented of file formats. Now many scanners can do this, and detect and remove viruses elegantly and quickly from *Word* documents.

Not so long ago we were still awaiting the next hit, which was bound to be an *Excel* macro virus; nevertheless, the appearance of Laroux in the wild shocked many anti-virus researchers. Detecting the Laroux virus presents a much more complex problem than detecting the *Word* viruses, as the *Excel* internal binary format is more complicated. The parsing procedures have to manipulate different tables of information, different sequences of pointers, and different data formats.

These new problems have initiated a new wave of anti-virus activity around the world: at present, the disinfection mechanisms are still under construction. At the moment, there is no standard method of disinfecting *Excel* spreadsheets.

The conclusion? Viruses will not die, nor will the anti-virus industry. Users will not be properly protected. *Word* and *Excel* viruses are just the current chapter in this never-ending story.

Languages

Two products from one company, one 'office', but each with a completely different macro language. The different development paths of *Word* and *Excel* are all too clear to those who have played extensively with the two applications' respective programming languages.

Any programming language built in to an application intended for manipulating documents and allowing automated document and data processing must clearly have access to the application's internal data. In the *Office* suite, there are two methods to access internal data: using functions and procedures, and using object-oriented programming.

Both languages discussed here (*WordBasic* and *Excel VisualBasic*) have the same parent, Basic, but use completely different methods for accessing the application's internal data. All internal objects in *WordBasic* are accessed by functions and simple statements. For example, this statement would modify the current style's font attributes:

```
FormatDefineStyleFont.Points = "12", .Bold = 1
```

Statements are extensions of normal Basic, and represent procedures with named arguments. In contrast, *Excel VisualBasic* uses an object-oriented method of access. All internal data is organized into an object hierarchy, and each object has its own methods and properties. The root object in this hierarchy is referred to as 'Application'. The following commands set the font attributes for the object 'myObject':

```
myObject.Font.Bold = True
myObject.Font.Size = 12
```

Whilst the statements given above for the two languages may appear remarkably similar, they actually function in a completely different way.

Another big difference between the two is the ability to use user-defined named constants in macros, a feature present only in *Excel VisualBasic*. Both languages can invoke external routines stored in a *Windows* dynamic-link library (DLL). This feature allows the programmer access to all system resources via the *Windows* API, offering huge flexibility and power, with similarly-proportioned risks.

Editing and Hiding

Excel offers an enhanced environment for source-code editing; it provides real-time syntax highlighting, and checks each line of code as it is typed for syntax errors. By contrast, *Word* only checks the syntax of a macro whilst it is being executed.

Both languages can make the source code for macros inaccessible to the user; *WordBasic* achieves this by setting the 'Execute Only' flag whilst the macros are being copied, whereas in *Excel* the same feat is accomplished by setting the sheet's 'Visible' property to 'xlVeryHidden'.

Documents, Templates, Sheets, Workbooks…

Only *Word* templates can contain *WordBasic* macros. A *WordBasic* macro is a set of functions and procedures – one of the procedures must be called MAIN, and will be executed when the macro is invoked. MAIN, like all other functions and procedures, can of course call functions and procedures from any macro in any loaded template. It is possible to create procedures and functions within a macro which are only accessible by other macros, not by the user.

Any *Excel* file can contain any number of macro sheets, each of which can contain any number of procedures/functions. Operations with macros from macro level are valid only for the macro sheet as a whole.

Macro Activation: Executing the Victim

Both *Word* and *Excel* have the unfortunate ability to run macros automatically on specified events. The first method by which this can be done is identical on both systems. By giving a macro a special name, the application can run it automatically when a user performs an operation such as opening/closing a document. *Word* and *Excel* recognize the following names as automatic macros; the now-infamous 'auto' macros:

Event	Word	Excel
Open a document	AutoOpen	Auto_Open
Close a document	AutoClose	Auto_Close
Application start	AutoExec	-
Application quit	AutoExit	-
Create a document	AutoNew	-
Activate a sheet	-	Auto_Activate
Deactivate a sheet	-	Auto_Deactivate

Another method of macro activation provided by *WordBasic* is the interception (or 'hooking') of built-in commands. By giving a macro the same name as a *Word* built-in command (for example, FileSave or ToolsMacro), *Word* will run it instead of the original command. For example, if a macro called FileOpen has been installed, it will be executed when the user selects the Open item from the File menu, or when he presses the Open button on the toolbar. Also, a programmer has the ability to determine the name of the command or macro assigned to a menu item or toolbar button – that is to say, he can modify the Open button to have a completely different purpose, including calling a custom macro.

The third method of activation is via the OnTime statement. For example, this command would run a macro called 'WakeUp' at 10:00:

```
OnTime "10:00", "WakeUp"
```

At any given time, only one macro can await execution – the scheduling is lost if *Word* is closed before the given time. In addition, the timer is not reactivated when *Word* is restarted.

Excel has a more complex and convenient system for processing events; it is possible to attach a macro to most *Excel* objects to allow event processing on that object. This information is accessible only at the macro level and is saved with the document.

The name of the event-processing macro is a property of many *Excel* objects, and macros can read and write to it. This table describes some properties and methods related to event processing:

Property/method	Applies to	Event description
OnAction	most visible objects	object is clicked
OnKey	Application	particular key/key combination pressed
OnTime	Application	specified future time
OnData	Application, Worksheet	DDE- or OLE-linked data arrives in *Excel*
OnDoubleClick	most visible objects	object double-clicked
OnSheetActivate	Application, Workbook, Worksheet...	object activates
OnSheetDeactivate	Application, Workbook, Worksheet...	object deactivates

Undocumented Documents: Going Inside…

Both applications save their documents in the OLE2 (Object Linking and Embedding) file format, a complex file system with directories and files (streams) which will not be described here. *Word* templates (remember, only a template can contain macros) are held in the OLE stream named 'WordDocument' within the file.

This stream contains all the information placed in the template by editing – including text, macros, toolbars, menus and styles. A pointer to the template area is stored at offset 118h from the beginning of the stream (not the beginning of the file!).

The template area consists of multiple variable-length records, each of which begin with signature bytes. A signature of 01h means that this record is a macro table. The macro table is further subdivided into records, each of which contains the offset of the macro from the beginning of the OLE stream.

If the OLE2 file contains an *Excel* file, things are more complicated: the OLE2 directory VBA_PROJECT contains all streams related to macros in an *Excel* document. It consists of one stream named 'dir' and at least one macro sheet stream.

The 'dir' stream contains references to object libraries, and objects called the 'small macro sheet table', the 'macro sheet table' and the 'global name table'. The 'macro sheet table' describes all the macro sheets: each record in this table contains the name of the OLE2 stream containing the macro sheet, and an offset to its name in the global name table.

The 'global name table' is a set of variable-size (10 or 12 bytes long) records. Each record describes one name which is used somewhere in the macros within the document, and each contains a pointer into an array of strings. Every name used in any macro is described somewhere in this name table.

Each macro sheet has a corresponding macro stream. The structure of this stream is:

```
header
static area
macro area
    line descriptor table
    macro body
```

The 'static area' consists of variable-size records. Each record can describe a declared variable, constant, function or procedure. References to the static area used in some statements (for example, Dim and Sub). The 'line descriptor table' contains each line of source code (with the line indent) and the offset to the compiled code for that line in the macro body, and a flag marking it as executable.

Code Representation

WordBasic uses a simple coding scheme to convert the macro source code into byte code by tokenizing. The usual form of a *WordBasic* token is a one-byte prefix code, which is followed by a variable amount of data relating to that prefix code.

The prefix represents Basic keywords such as 'If' or 'While', in addition to language constructs such as user-defined names, labels, internal function calls, and statements. Below is a list of some of these special prefixes:

Prefix	Optional data	Description
0x51	none	space
0x52	none	tab
0x64	none	new line
0x65	string	alphanumeric label
0x66	word	integer label
0x67	word	internal function name
0x68	8 bytes	double integer constant
0x69	string	name
0x6A	string	string constant
0x6B	string	comment (with ')
0x6C	word	integer constant
0x6E	byte	several spaces
0x6F	byte	several tabs
0x70	string	comment (with REM)
0x73	word	named argument of statement

Excel VisualBasic uses partially compiled code, which is intended for direct execution on a stack machine, in the same manner as Forth. This method is faster on execution, but significantly slower on editing, than the method

WordBasic uses. *Excel* compiles the macro code line-by-line while the macro is edited. Each line of source code is compiled into a set of micro commands for execution by the stack machine.

Each micro command consists of a two-byte command identifier followed by optional data aligned on a two-byte boundary. To illustrate this, the diagram at right reproduces the decompilation process of one line of *VisualBasic for Applications* (*VBA*) code.

Each micro command controls both the token representation and the stack machine. The stack machine is controlled through two pseudo-commands:

- push – put decoded token onto stack
- pop – get token from stack

Careful analysis obtains a strange result – the micro command ‘pop’ does not only get tokens from the *top* of stack! I have no words for the language creators…

Not all names in a macro are contained in the global name table. Micro commands can include Basic keywords and some internal Basic functions such as Format or Error.

```
0000003C:  00A3 0001 == push 1
      00A3 - opcode 'push integer'
      0001 - constant value
Stack: 1
00000040:  00A3 0002 == push 2
      00A3 - opcode 'push integer'
      0002 - constant value
Stack: 2
        1
00000044:  00AD 0006 == “sheet1” push
      00AD - opcode 'push string'
      0006 - constant length
Stack: "sheet1"
        2
        1
0000004E:  00AD 0005 == “book1” push
      00AD - string constant
      0005 - constant length
Stack: "book1"
        "sheet1"
        2
        1
00000058:  0024 0782 0001 == Workbooks(pop arg)push
      0024 - name(arguments)
      0782 - pointer into global name table
      0001 - number of arguments
Stack: Workbooks(“book1”)
        "sheet1"
        2
        1
0000005E:  0025 078C 0001 == pop.Worksheets(pop arg) push
      0025 - pop.name(arguments)
      078C - offset to name in global name table
      0001 - number of arguments
Stack: Workbooks(“book1”).Worksheets(“sheet1”)
        2
        1
00000064:  0025 0798 0002 = pop.Cells(pop 2 args) push
      0025 - pop.name(arguments)
      0798 - offset to name in global name table
      0002 - arguments count
Stack: Workbooks(“book1”).Worksheets(“sheet1”).Cells(1,  2)
0000006A:  0020 05A4 == push n
      0020 - name
      05A4 - pointer into global name table
Stack: n
        Workbooks(“book1”).Worksheets(“sheet1”).Cells(1,  2)
0000006E:  000B == pop + pop push
      000B - pop plus pop
Stack: Workbooks(“book1”).Worksheets(“sheet1”).Cells(1, 2) + n
00000070:  0027 0194 == a = pop
      0027 - name = stack; end decode
      0194 - pointer into global name table
Stack: none
Result: a = Workbooks(“book1”).Worksheets(“sheet1”).Cells(1, 2) + n
```

What’s Next?

In version 5.0, *Excel* acquired *VisualBasic*, as well as the *Excel 4.0* macroing language. Both types of macro sheets are supported in *Excel 5.0* and later. *WordBasic* was changed in *Word 6*, and macros from previous versions must be converted (automatically or manually) before use.

These modifications did not affect the two products equally. *Excel* has a more convenient, professional and powerful language, the next version of which (*VBA5*) will be the standard application language in *Office 97*. An analysis of *PowerPoint* data files showed the presence of *VisualBasic* macros in those files.

Unfortunately, *Excel* and *Word* are not the only applications which make it possible to create macro viruses. *AmiPro* also has macros, and one virus has been written for that system [*Green_Stripe; see VB, March 1996, p.11*]; however, *AmiPro* documents are not widely exchanged.

Do other systems exist that will allow the easy creation and subsequent widespread replication of yet more brand new viruses and virus types?