SCRIPT-BASED MOBILE THREATS

Mark Kennedy
Symantec (SARC Division), 2500 Broadway, Suite 200, Santa Monica, CA 90404-3036, USA
Tel +1 310 453 4600 • Fax +1 310 453 0636 • Email mkennedy@symantec.com

ABSTRACT

Microsoft *has created a very flexible, powerful environment on its Win32 platforms. The combination of simple scripting languages coupled with powerful objects through the common interface of COM makes it possible for relatively unsophisticated programmers to create fully functional business applications. Moreover,* Microsoft *has extended these tools to run from HTML, making deployment of these applications very inexpensive.*

The down-side to this power and flexibility is that it is now possible for malicious people to utilize this same technology to attack machines. Exacerbating this threat are standard HTML-based email programs that will execute these scripts. This allows the perpetrator to deliver his package anonymously, and for that package to propagate utilizing the victim's own email address book.

To combat this threat we can inject an intelligent layer, a script firewall if you will, to determine which scripts are allowed to execute. This layer can be customized to the individual or organization to balance the business requirements against the security implications. This paper will explore the difficulties of building a script behaviour blocking system and examine how effective such a system is against today's malicious threats.

WINDOWS SCRIPTING HOST: MYTHS AND REALITIES

Anyone not living under a rock for the past few months knows that the hot topic, at least as far as the media is concerned, is script-based worms. Below are a number of clippings regarding some of the more infamous examples.

> *Copycat viruses following 'ILOVEYOU' computer bug are no joke*
> 4 May, 2000
> (*CNN*) Hours after the self-propagating and destructive 'ILOVEYOU' virus destroyed critical files and jammed countless electronic mail systems, computer network administrators battled at least one copycat virus dubbed 'very funny.' 'There'll be hundreds of these maybe thousands.'
>
> *New computer virus more destructive, but appears less infectious*
> 19 May, 2000
> (*CNN*) 'Otherwise, it's totally new code. But there's a common idea.' Rather than the same subject line each time, 'NewLove' is polymorphic. Each time, it takes the name of a recently accessed file on the user's machine and uses that name, along with 'FW:'. This can work much better than 'ILOVEYOU,' because users can't be on the lookout for a specific subject line. Instead, the subject line may be a file name that is trusted – especially among co-workers. 'It's really quite clever.'
>
> *Viruses boom on the Net*
> 18 January, 2000
> by Mark Leon
> (*CNN*) The reasons for concern do not stop there. Not only do the unscrupulous have a bigger field to play in, they also have tools that are easier to use and potentially more dangerous.
> 'The advent of macro and script viruses – viruses written in Macro languages such as Word Macro and VBScript – makes it fairly easy to write new ones.' 'This is mobile code. As it becomes easier to use, we will see more mobile virus code.'
>
> *'Bubbleboy' email virus benign, but also a warning*
> By John Fontana
> *Network World*, 15 November, 1999
> The recently discovered 'Bubbleboy' email virus probably won't be causing any headaches for IT executives, but they better ready their defenses for his offspring. Bubbleboy variants will have the ability to spread faster than the Melissa virus, which ate up corporate email systems earlier this year. The most frightening aspect of the new worm, researchers say, is that users do not have to open an attachment to activate it, which was the trigger for Melissa.

The press and others have grossly misreported the techniques and susceptibilities of these script-based attacks. There are two scripting languages widely available on today's *Windows* machines: VBScript and JavaScript. Both are nearly identical in power and syntax. A program written in one can be translated to the other in a matter of hours. The *Windows Scripting Host* (*WSH*) determines which language a piece of script is written in and calls the appropriate interpreter.

The most important thing to understand is that by itself the *WSH* is totally incapable of doing harm. The primitives of the languages (both VBScript and JavaScript) have no ability to affect the File System, the Registry, or *Outlook*. They do, however, have the ability to leverage COM objects and ActiveX controls residing on the system and through them wreak havoc.

To understand worm behaviour, there are two basic aspects to keep in mind:

- *Arrival*: the means by which the worm activates on the victim's machine.
- *Departure*: how the worm moves on.

Arrival is accomplished when the script code is executed on the victim's machine. Script can be hosted in a number of forms and can be activated in a number of ways, some automatic.

Departure has mainly been via *Outlook's* automation capability (JS/Kak and VBS/Network are exceptions to this). The *Outlook* object can enumerate the *Outlook* address book and send email, including email with attachments. The worms are able to spread by taking advantage of this object.

Standalone script code such as VBS/LoveLetter comes in an attachment. This file, by means of its extension will, when launched, cause the script to be executed. Standalone script may be 'encoded' by utilizing a tool from *Microsoft*. Encoding turns the script into a quasi-encrypted binary form that the scripting engine can reverse. This is a simple form of obfuscation. Standalone script requires that the victim launch the attachment or file.

Embedded script is script code that is embedded within HTML. This script code can be located inside a Web page or as the body of an HTML email message. This script code will execute the moment it is viewed. Thus, script code arriving in an HTML email message will execute if the victim merely views the message in a preview pane. The victim will have no idea that code has executed other than possibly a security popup message from the reader. Embedded script can also be 'encoded' using the same technique described previously. More recent revisions of *Outlook Express* have disabled all script objects from within email bodies.

In the recent outbreaks, the media has continually expressed the mistaken belief that only people running *Outlook* were at risk. Nothing could be further from the truth. *Outlook* is the mechanism that allows some of the worms to spread. Any email program that could launch attachments would have been sufficient to deliver the payload (in VBS/NewLove's case complete destruction of the machine). For example, *Lotus Notes* has an option to launch attachments. It was not important how the worm was launched only that the worm was launched.

Script-based worms are effective, easy to write, and occasionally deadly. They are usually followed by a slew of variants due to the fact that they are delivered with source code. Due to the ready availability of sample code and the simplicity of the languages little skill is required to generate the original, and virtually none for the knock offs. As a matter of fact, virtually all of the major worms have at their core sample code which can be located on the *Microsoft* Web site.

Since scripts can be executed from a wide array of formats, the virus/worm writer can choose his method of delivery. Some do not even require the victim to do anything more than preview the host message. From standalone .VBS files to embedded HTML to OLE compound documents the avenues of ingress for script seem almost limitless. This multitude of paths seriously compromises the scanning-based methods of dealing with potential threats. Almost any file can contain executable script.

Choose Your Playing Field

> *'By the end of last year, there were more than 200 million PCs connected to the Internet. Ninety percent of these are Windows machines running the same applications, such as Word, Microsoft Exchange, and Excel. For the first time, we have a computing monoculture. Monocultures in the natural world are extremely vulnerable to pests such as viruses.'*
>
> Carey Nachenberg, Chief Researcher at *Symantec's Anti-virus Research Center*

Script can utilize many objects on *Windows* machines. Some are part of the operating system and provide a known arena in which to play. Others are prevalent enough to allow for wide coverage. The virus/worm writer can choose which of these objects to use and, thus, determine how wide an audience to attack.

The homogeneous *Windows* machine makes it fairly easy to predict how a worm written on one machine will behave on another. The more generically written the worm, the better it will spread. Scripting even provides functions to abstract out the differences in directory structure that might be encountered. Some virus/worm writers have utilized this capability and, in the process, have written code that is more robust than some 'professional' code.

When *Microsoft* made the decision to allow *Windows* to be scriptable, they foresaw these variations in installations. They wanted to provide a powerful, flexible ability to automate certain tasks. Additionally, they wanted to provide this flexibility in their *Office* applications. *Microsoft* achieved its goal. Simple scripts can be written by non-sophisticated users to accomplish a wide variety of tasks.

In their never-ending quest for flexibility, *Microsoft* allowed these same scripting engines to be hosted in HTML. This allows scripts to execute via Web pages and HTML email. Either by design or oversight there is no distinction between script running in a local source file and script running in a spam email. What started as a tool for a local administrator has now become portal for the unscrupulous.

Script can also be embedded in OLE compound documents. Worms have been attached to help files with this method. VBS/Stages.A is a new generation worm that takes advantages of a number of quirks in the *Windows* operating system. For example, *Windows* will show the extension .SHS despite the user choosing to display all extensions.

OBJECTS: THE SOURCE OF THE POWER

Fortunately, the base scripting languages do not have the ability to affect the system on their own. They have no persistent storage. In order to reach out and manipulate the operating system they must make use of objects. Most of these objects were provided for just this purpose. The number of the objects is, however, limited. These limitations provide the necessary choke-point that allows them to be blocked before they can do any damage.

Before the worm can interact with the system, it must create an instance of the object. It then calls methods in that object to attempt its task. If that object does not comply with what the worm wants done, there is no recourse. The worm is stopped and rendered harmless. Since most scripts that reside on Web pages do not attempt to make use of these objects (at least not legitimate script), restricting access to these objects poses few problems.

The objects most used by worms are the Scripting.FileSystemObject (FSO) object, the WScript.Shell object, the WScript.Network, and the Outlook.Application. The first three exist on all machines running the *WSH*. The Outlook.Application object is only present if the target machine has *Outlook* installed. This reduces the number of viable targets, but facilitates easy spreading.

Assisting the virus writer in his work are the numerous example programs and code snippets provided by *Microsoft*. A few minutes perusing their Web site or scanning the examples provided with the operating system will yield virtually all of the framework used by the various script-based worms in the wild today.

The Scripting.FileSystemObject

The Scripting.FileSystemObject provides complete access to the file system. It consists of powerful methods for manipulating the file system, including methods to abstract out those portions that might vary among *Windows* installations, such as the *Windows* directory. A subset of the methods provided is given in the list below. All of the major script-based worms make use of this object. Minimally, it is used to place a copy of itself in some specific location so it would be run on system startup. Some use it to do damage.

The GetSpecialFolder method is the means to allow for variations in system installation. It returns the *Windows* folder (generally C:\Windows or C:\Winnt), the System folder (generally C:\Windows\System or C:\Winnt\System32) or the temporary folder. The first two directories are always on the *Windows* path (meaning a non-qualified launch will always find them). Additionally, users are not likely to examine these folders. In fact, newer versions of *Windows* actually prevent the user from examining either of these options unless they bypass a somewhat intimidating warning. This, coupled an official sounding name for the worm, such as WIN32DLL.VBS, insures that only the brave of heart would dare remove it. A reference to the partial Scripting FileSystem Object is provided below.[1]

Language Element	Description
BuildPath	Appends a name to an existing path.
Close	Closes an open TextStream file.
CopyFile	Copies one or more files from one location to another.
CopyFolder	Recursively copies a folder from one location to another.
CreateFolder	Creates a folder.
CreateTextFile	Creates a specified file name and returns a TextStream object that can be used to read from or write to the file.
DeleteFile	Deletes a specified file.
DeleteFolder	Deletes a specified folder and its contents.
DriveExists	Returns True if the specified drive exists; False if it does not.
FileExists	Returns True if a specified file exists; False if it does not.
FolderExists	Returns True if a specified folder exists; False if it does not.
GetAbsolutePathName	Returns a complete and unambiguous path from a provided path specification.
GetBaseName	Returns a string containing the base name of the file (less any file

	extension), or folder in a provided path specification.
GetDrive	Returns a Drive object corresponding to the drive in a specified path.
GetExtensionName	Returns a string containing the extension name for the last component in a path.
GetFileName	Returns the last file name or folder of a specified path that is not part of the drive specification.
GetFolder	Returns a Folder object corresponding to the folder in a specified path.
GetSpecialFolder	Returns the special folder specified.
GetTempName	Returns a randomly generated temporary file or folder name.
Move	Moves a specified file or folder from one location to another.
MoveFile	Moves one or more files from one location to another.
MoveFolder	Moves one or more folders from one location to another.
OpenTextFile	Opens a specified file and returns a TextStream object that can be used to read from, write to, or append to the file.
Read	Reads a specified number of characters from a TextStream file and returns the resulting string.
ReadAll	Reads an entire TextStream file and returns the resulting string.
ReadLine	Reads an entire line (up to, but not including, the newline character) from a TextStream file and returns the resulting string.
Remove	Removes a key, item pair from a Dictionary object.
RemoveAll	Removes all key, item pairs from a Dictionary object.
Skip	Skips a specified number of characters when reading a TextStream file.
SkipLine	Skips the next line when reading a TextStream file.
Write	Writes a specified string to a TextStream file.
WriteLine	Writes a specified string and newline character to a TextStream file.

There is one important note to the FSO and the files it reads, writes, or creates: they need not be text-based. Although the documentation and the examples show them as being 'Text' they are only strings. Strings in script can consist of binary values and as such can be used to read and write binary files. By this means, a script can drop and then execute an EXE file. Scripts have moved into the world of EXE-based worms and/or viruses. It should be noted that script carrying the ASCII-based form of an executable would be significantly larger than that executable. In low bandwidth areas this may provide a significant impediment to spreading.

The WScript.Shell Object

We have seen that the FSO has laid bare the file system of *Windows*-based machines. This leaves out the other piece of the *Windows* system: the Registry. Not to worry. *Microsoft* added the WScript.Shell object – for which a subset of the methods provided is shown below[2].

Language Element	Description
ConnectObject	Connects an objects event sources to functions with a given prefix.
CreateObject	Creates an object specified by the strProgID parameter.
CreateShortcut	Creates an object reference to a shortcut or URLshortcut.
ExpandEnvironmentStrings	Expands the process environment variable and returns the result string.

RegDelete	Deletes from the Registry the key or value named strName.
RegRead	Returns the Registry key or value named by strName.
RegWrite	Sets the Registry key or value named by strName.
Remove	Deletes the environment variable specified by strName.
Run	Creates a new process that executes strCommand.
Save	Saves a shortcut to the specified location.
SendKeys	Sends one or more keystrokes to the active window as if typed at the keyboard.
Sleep	Places the script process into an inactive state for the number of milliseconds specified and then continues execution.
SpecialFolders	Accesses the *Windows* shell folders such as the desktop folder, the Start menu folder, and the personal document folder.

With WScript.Shell, the virus/worm writer has access to the *Windows* Registry – another place where most users fear to tread. By setting an entry into the HKLM\Software\Microsoft\Windows\CurrentVersion\Run key, the worm can ensure it gets run on each boot. Again, a sufficiently frightening name prevents users from considering deleting it.

The Run method launches any program. It can even launch a program in a hidden state.

The SendKeys method can be used to drive the application once started. This could be used to mask suspicious behaviour by driving a proper application to do bad things. Examples of this have been seen in the macro virus world. Some macro viruses have spread by sending the keyboard commands to cut and paste themselves from one document to another. This is done to defeat heuristic detection of macro copy.

If destruction is the intent then the Run, RegWrite, and RegDelete methods answer the call. Running a 'format' or 'deltree' command wipes out masses of files. Deleting or modifying important Registry keys can render a system just as useless.

Some worms have used the RegWrite method childishly to set the *Internet Explorer* home page to a sex site or otherwise deface the victim's machine.

Access to network drives gives a stealthy avenue to spreading. Mapped drives can be examined to see if they appear to be other boot drives. If so, placing a copy of the worm in the appropriate folder will cause it to be launched when that machine next reboots. This is precisely what VBS/Network does.

The SpecialFolders object is used to look for file names off the user's system. By checking the 'Recent' location you can get the name of a real document the user recently accessed. VBS/NewLove used a flawed version of this to create its new subject line and attachment name. This meant that, potentially, every copy of VBS/NewLove could have had a different subject line and attachment name, making it difficult to come up with mechanism to recognize the attack.

The WScript.Network Object

There is an additional object specifically for dealing with the network. This object allows mapping arbitrary network drives as well, as enumerating the drives that are already mapped. Since the format of the network drive is in UNC, any IP address could be attempted. Using this

method, a worm could spread without the aid of an email program. The VBS/Network worm does just that. Based on the sample file included with *Windows 98* and *WSH*, it seeks out network drives and infects them. A partial list of WScript.Network object methods is shown below[3].

Language Element	Description
EnumNetworkDrives	Returns the current network drive mappings.
MapNetworkDrive	Maps the share point specified by strRemoteName to the local resource name strLocalName.
RemoveNetworkDrive	Removes the current resource connection denoted by strName.

Outlook As Transportation Mechanism

Why is *Outlook* such a popular target? The answer to this is three-fold. First, there are many examples of how to drive *Outlook* and how it works. As mentioned before, the source to these worms is distributed, making cutting and pasting simple (one EXE-based worm even used script to send itself because the virus/worm writer did not know COM and using script was easy).

The second reason is *Outlook* is easy for virus/worm writers to acquire. Conversely, *Lotus Notes*, which could be driven via agents, requires a certain knowledge of *Notes* and access to the program. Most virus/worm writers do not have *Notes* and a *Notes* server running on their local machines. Thus, it is more difficult to write and test a worm that makes use of *Notes*.

Lastly, *Outlook* provides powerful methods via scripting. You can enumerate the address book (or books) to find email addresses. You can enumerate the Inbox to find out who has sent email and return mail to them. You can enumerate the Sent box and provide follow-up. All of these answer a key need of the worm: how do I find new targets to infect? *Outlook* provides a ready set of valid email addresses. It provides valid and varying subject lines. In short, it offers both the target's name and a modifiable method for looking normal – all to entice the next victim into opening the mail or attachment and, thereby, spreading the worm.

Microsoft provides excellent examples of how to drive *Outlook* via scripting and Visual Basic for Applications (VBA), the language *Office* application's macros are written in. Figure 1 is the example *Microsoft* provides for enumerating the *Outlook* address book. This is virtually identical to the code found in *Outlook* replicating worms, including W97M/Melissa.

```
Sub RetrievePAB()
 Dim aPAB() As Variant
 Dim adl As Outlook.AddressList
 Dim e As Outlook.AddressEntry
 Dim i As Integer
 ReDim aPAB(100, 2)
 Set nsMAPI = ol.GetNamespace('MAPI')
 'Return the personal address book.
 Set adl = nsMAPI.AddressLists('Personal Address Book')
 'Loop through all entries in the PAB
 ' and fill an array with some properties.
 For Each e In adl.AddressEntries
 'Display name in address book.
 aPAB(i, 0) = e.Name
 'Actual email address
 aPAB(i, 1) = e.Address
```

```
 'Type of address ie. internet, CCMail, etc.
 aPAB(i, 2) = e.Type
 i = i + 1
 Next
 ReDim aPAB(i - 1, 2)
End Sub
```

Figure 1: Microsoft's sample code for enumerating the Outlook address book[4]

The Scriptlet.Typelib Object

The scriptlet object was intended to generate type libraries for *Windows* script components. The BubbleBoy author discovered that the object could be used to create arbitrary content in an arbitrary file. A partial list of the methods provided by this object is shown below[5].

Language Element	Description
Path	The file name for the type library, which can optionally include a path.
Doc	A string containing any information that is stored in the Registry with the type library information.
Write	Write the current type library.

CASE STUDIES

Now that the main players have been introduced, we can take a look at the role these objects have played in some of the most infamous worms. We will also examine some of the variants and how easily they were constructed.

VBS/BubbleBoy

> *WARNING: If you get an E-mail titled: 'Win A Holiday'. DO NOT open it. Delete it immediately. Microsoft just announced yesterday. It is a malicious virus that WILL ERASE YOUR HARD DRIVE.*
>
> *At this time there is no remedy. Forward this to everyone IMMEDIATELY!!*
>
> *SARC* Hoax Database

Prior to November 1999, messages about viruses that spread merely by opening the message were urban legends. VBS/BubbleBoy changed that. By making use of both a security hole and a design flaw in the Scriptlet.TypeLib object, VBS/BubbleBoy was able to create an arbitrary file in an arbitrary directory. Then, by leveraging *Microsoft's* own security policy regarding scriptable objects it was able to activate objects without the warning that would ordinarily be generated. The ramifications of this were staggering. For the first time an email message could infect a machine simply by being read or, for that matter, viewed in the preview pane. This development turned the hoax above into reality.

How did this happen? *Microsoft* uses safety bits to stop ActiveX objects from being used when launched from sources outside the local machine. When code comes via an external Web page or a HTML email, objects used must be tagged as 'safe for scripting'. If not, a warning dialog is

displayed to alert the user to the possible ramifications of proceeding. Only objects of specific and limited use should carry this tag. VBS/BubbleBoy used HTML email to execute a supposedly innocuous ActiveX object that was marked safe for scripting. This object was intended to make type libraries for objects. But it could do more. The Scriptlet.TypeLib object did not check that it was creating a type library. VBS/BubbleBoy used it to drop a .HTA (HyperText Application) file in the *Windows* startup folder. Upon the next boot, this file would execute. Figure 2 shows the relevant content of the email.

```
<html>
<body alink='#ffffff' bgcolor='#000000' link='#ffffff'
text='#ffffff' vlink='#ffffff'>
<object classid='clsid:06290BD5-48AA-11D2-8432-006008C3FBFC'
id='Vandelay'>
</object>
<script language='VBScript'>
. . . BubbleBoy Worm Code . . .
```

Figure 2: VBS/BubbleBoy's use of Scriptlet.object

The email dropped its file into the C:\Windows\Start Menu\Programs\StartUp directory. While this directory would not exist on all machines, it was sufficiently generic. The script could not query the FSO to find the *Windows* directory because that would have set off alarms. When executed from the startup folder, this file was operating not from the Internet security zone, but rather from the local security zone. *Microsoft* clearly presumed if it is on your local machine, it can do what it wants without any warnings. VBS/BubbleBoy could now make full use of the other objects so spread itself.

VBS/BubbleBoy used the Scripting.FileSystemObject, the WScript.Shell, and the *Outlook* object. This is a fairly typical mix for worms. Figure 3 shows some of the uses of the WScript.Shell object. The owner and organization of the machine are changed and it checks to see if it has run before. These changes are the only lasting effect of the worm.

```
Set Jerry = CreateObject('WScript.Shell')
Jerry.RegWrite
'HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\
RegisteredOwner','BubbleBoy'
```

Figure 3: VBS/BubbleBoy's use of the Scripting.FileSystemObject

Figure 4 shows the use of the FSO. It processes through itself and removes the extraneous text added as a side effect of the Scriptlet.TypeLib object. It then transforms the text for inclusion in the HTML of the email it will send to spread itself.

```
Set Jason = CreateObject('Scripting.FileSystemObject')
Set Michael = Jason.OpenTextFile(Replace(Replace(Replace(Location.
PathName,'/',''),'%20',' '),'%FA','Ú'),1)
```

Figure 4: VBS/BubbleBoy's usage of the Scripting.FileSystemObject

Lastly, Figure 5 shows the totally typical *Outlook* loop. The only variation here is that one message is sent to everyone in the address book. Others have sent individual emails to each entry in the address book (this mass mailing is what can crash mail servers).

```
Set Elaine = CreateObject('Outlook.Application')
Set Seinfeld = Elaine.GetNameSpace('MAPI')
For Each Cosmo In Seinfeld.AddressLists
If Cosmo.AddressEntries.Count <> 0 Then
Set Kramer = Elaine.CreateItem(0)
. . .
Kramer.Send
```

Figure 5: VBS/BubbleBoy's usage of the Outlook.Application object

VBS/Network

VBS/Network is a self-spreading non-destructive worm that has the ability to infect a machine simply because it is connected to the Internet with a non-protected share. It attempts to map random IP addresses' C share onto drive J using the WScript.Network object. It then copies itself to various locations on that machine in the hope that this will continue to spread the infection.

Figure 6 shows the usage of the network object. A random sharename, in the form \\w.x.y.z\c is created and the map attempted. Then all network drives are enumerated looking for one that has that sharename. Then the infection spreads to successfully mapped drives using the FSO, as shown in Figure 7.

```
set wshnetwork = wscript.createobject('wscript.network')
wshnetwork.mapnetworkdrive 'j:', sharename
```

Figure 6: VBS/Network's usage of the WScript.Network object

```
Set fso = CreateObject('scripting.filesystemobject')
fso.copyfile 'c:\network.vbs', 'j:\'
```

Figure 7: VBS/Network's usage of the Scripting.FileSystemObject

The worm attempts to copy the file to seven different locations. It is also keeping track of machines it has infected in a log on the host. The CreateObject function is flexible. It takes a case-insensitive string. Since the string could be a variable assigned over several statements, it is not possible to see which object was being created from a static evaluation of the file. Sometimes the brute force approach does work. Since its discovery in February, there have been over 1,500 submissions[6]. It is somewhat astounding that IP addresses with unprotected shares named C would be found by random chance and be capable of reaching these numbers.

JS/Kak

At the time of writing, JS/Kak is the most prevalent ItW worm. Like VBS/BubbleBoy, it spreads by the mere viewing of an infected HTML email. What is different is that it hides at the end of each email its victim sends out. It accomplishes this by replacing the *Outlook Express* (not *Outlook*) signature file with a copy of itself. This causes every HTML email sent from an infected machine to in turn infect a recipient who views it as HTML. The worm sets itself to run twice during startup, once in the HKEY_LOCAL_MACHINE\Software\Microsoft\Windows\CurrentVersion\Run key and once in the startup folder.

Figure 8 shows the unique way in which JS/Kak invokes the WScript.Shell object. Since JS/Kak runs as an HyperText Application (HTA), it utilizes the Object tag and GUID (Globally Unique IDentifier) to invoke the object. This is equivalent to the 'var wsh = new ActiveXObject('WScript.Shell');' statement in JavaScript. Also unusual is the fact it is written in JavaScript. Notice also the use of the Run method to set the Registry. This is interesting since the WScript.Shell object has a method to set the Registry directly.

```
<object id='wsh' classid='clsid:F935DC22-1CF0-11D0-ADB9-
00C04FD58A0B'>
</object>
<SCRIPT>
wsh.Run(wd+'Regedit.exe -s '+wd+'kak.reg');
```

Figure 8: JS/Kak's use of the WScript.Shell object

Figure 9 shows the use of the FSO. Here, again, we see functionality of the object overlooked. 'wd' is set to C:\Windows instead of interrogating the FSO's GetSpecialFolder method to return it. This mistake alone is likely to prevent the worm from functioning on most *Windows NT* systems. But for all its shortcomings, JS/Kak has remained a durable beast. It still exists in the wild, despite being discovered in December of last year. In fact, the number of infections is growing with 141 submissions in March 2000, 4813 in May, and already 2641 by mid June[7].

```
fs=new ActiveXObject('Scripting.FileSystemObject');
fl=fs.GetFolder(wd+'Applic'1\\Identities');
sbf=fl.SubFolders;
. . .
wd='C:\\Windows\\';
```

Figure 9: JS/Kak's use of the Scripting.FileSystemObject

VBS/LoveLetter

Not since Melissa has a piece of malware garnered more press coverage. Spreading faster than Melissa and crashing mail servers, VBS/LoveLetter captured the imagination of the press. Within 24 hours it had spawned over 20 variants, one particularly nasty one posing as a cleanup tool from *Symantec*. Claims of dollar damage followed, with one estimate of up to US$10 billion[8] being quoted. All of this, and the most innovative aspect of the worm, was the human engineering required to get people to launch an attachment.

Upon examination VBS/LoveLetter appears to have been pieced together from different scripts to form a hodgepodge of functionality. Part worm, part Trojan, part virus, it included *mIRC* infection and password stealing; it packed a lot of functionality into one package. It made more use of the FSO than any previous worm.

Let us break it down and look at the different ways the FSO is used. Figure 10 shows some of the standard behaviour we have seen before. Since the script is launched from an email program as an attachment it cannot know the file name it is running with. Script addresses this by providing the WScript.ScriptFullname method to retrieve the name of the script file currently running. Then, by using the FSO's ReadAll method, it pulls in its source code for later replication. It uses the GetSpecialFolder in dropping its copies so it will run on all systems.

```
Set fso = CreateObject('Scripting.FileSystemObject')
set file = fso.OpenTextFile(WScript.ScriptFullname,1)
vbscopy=file.ReadAll
Set dirwin = fso.GetSpecialFolder(0)
```

Figure 10: VBS/LoveLetter's standard use of the Scripting.FileSystemObject

Not content with simple replication, VBS/LoveLetter set out to seed copies of itself on all local and network drives. It did this by attempting to masquerade as some of the user's files. This constituted the destructive payload of the worm, since existing script files and .JPG files were overwritten. Interestingly, .MP3 files were preserved by setting the Hidden attribute bit. The virus/worm writer(s) did not mind overwriting porn but drew the line at pirated music. This bit of overkill made use of new elements of the FSO – drive enumeration, folder enumeration, and file enumeration allowed it to reach beyond the current machine.

VBS/LoveLetter also used a *mIRC* script to spread an HTML version of the worm. Where did this HTML version come from? With the FSO and native string processing routines, it is able to create an HTML version of itself. Figure 11 is the routine to do this. It is interesting to note this routine opens and reads the script again. This is despite the fact that a copy of the file is already read in (shown by Figure 10). This routine shows the powerful text-processing capabilities of VBScript. It also shows inefficient coding since the 'l1' variable is set, then the ubound(lines) is called for the loop!

```
set fso=CreateObject('Scripting.FileSystemObject')
set c=fso.OpenTextFile(WScript.ScriptFullName,1)
lines=Split(c.ReadAll,vbcrlf)
l1=ubound(lines)
for n=0 to ubound(lines)
lines(n)=replace(lines(n),'''',chr(91)+chr(45)+chr(91))
lines(n)=replace(lines(n),''''',chr(93)+chr(45)+chr(93))
lines(n)=replace(lines(n),'\',chr(37)+chr(45)+chr(37))
. . .
set b=fso.CreateTextFile(dirsystem+'\LOVE-LETTER-FOR-YOU.HTM')
set d=fso.OpenTextFile(dirsystem+'\LOVE-LETTER-FOR-YOU.HTM',2)
d.write dt5
```

Figure 11: VBS/LoveLetter HTML creation routine

One aspect of VBS/LoveLetter remains: spreading via the Outlook.Application object. This is a textbook example of the process. In Figure 12 we see the sending of the email with attachment. Since it sent copies to all entries in the address book (causing some mail servers to fail) it did try to avoid this on subsequent runs by using the WScript.Shell object to store, in the Registry, the number of copies sent. If the address book contains new entries then the whole book is enumerated again and a new batch of emails sent out.

```
set regedit=CreateObject('WScript.Shell')
set out=WScript.CreateObject('Outlook.Application')
set mapi=out.GetNameSpace('MAPI')
for ctrlists=1 to mapi.AddressLists.Count
. . .
male.Subject = 'ILOVEYOU'
male.Body = vbcrlf&'kindly check the attached LOVELETTER coming from
me.'
```

```
. . .
regedit.RegWrite
'HKEY_CURRENT_USER\Software\Microsoft\WAB\'&malead,1,'REG_DWORD'
```

Figure 12: VBS/LoveLetter's use of the Outlook. Application and WScript.Shell objects

VBS/LoveLetter Variants

At the time of this writing there have been 28 variants of the VBS/LoveLetter worm[9] making it the most copied script-based worm in history. There are a number of factors that led to this. It was heavily hyped in the media, as were the variants. It was widely distributed, so millions of people had access to the source. And modifications to the source were trivial to make. Some just translated the message text or the attachment name. As you can see in Figure 12 the message text and subject lines are just strings. Absolutely no skill is required to change these and send it on.

VBS/NewLove – Destruction Has A New Name

The majority of the script-based worms did little or no damage. This changed with the arrival of VBS/NewLove. The name is a bit of a misnomer since apart from its distribution technique it had nothing to do with VBS/LoveLetter. It was a .VBS file, used the same Outlook.Application techniques, and followed two weeks after VBS/LoveLetter, so the connection was made. However, examination of the code shows clearly that it was no variant. It is, in many ways, more sophisticated and innovative than VBS/LoveLetter. It is interesting to note that while the WScript.Shell object is created, it is never used. There is no need to use the Registry to execute the worm on the next run – there will be no next run. One is fatal.

VBS/NewLove is a polymorphic worm, varying not just the subject line and attachment name but the physical code itself. The first two are done by reaching in to each victim's recent document list, the last one by altering the bogus comment lines generated to change the attachment size and look. Due to a bug in this morphing code the file size grows by approximately 100 KB for each revision of the worm, allowing the generation of a given infection to be determined. Figure 13 shows what starts as a fairly standard initialization sequence. Then, however, we get the first portion of the polymorphism: the enumeration of the Windows\Recent folder looking for a random name. Now the subject line will be modified to something from the victim's machine and is consequently realistic. One flaw in this scheme, though, is the handling of the attachment's extension. A random extension is added, but the old extension is not removed. The combination is then followed by the obligatory .VBS resulting in an unlikely conglomeration like 'My Document.doc.jpg.vbs.'

```
Set fso = CreateObject('Scripting.FileSystemObject')
Set MySelf = fso.GetFile(WScript.ScriptFullName)
Set MyFile = fso.OpenTextFile(WScript.ScriptFullName,1)
WindowsDir = fso.GetSpecialFolder(0)
```

Figure 13: VBS/NewLove's use of the Scripting.FileSystemObject

VBS/NewLove's use of the Outlook.Application object is nearly identical to VBS/LoveLetter's – not surprising since all the steps are predictable and required. Figure 14 contains this code.

```
set Outlook=WScript.CreateObject('Outlook.Application')
set MAPI=Outlook.GetNameSpace('MAPI')
```

Figure 14: VBS/NewLove's use of the Outlook.Application object

VBS/Stages.A – The New Worm On The Block

VBS/Stages.A demonstrates precisely why the 'scan only' solution does not work. *Microsoft* has so integrated scripting into *Windows* that scanning software is like the proverbial Dutch boy plugging the many places in the dike. VBS/Stages.A uses a .SHS file (Shell Scrap) as its point of entry. This extension was not being scanned or blocked at the gateway allowing the files to enter. Then things got interesting.

Utilizing an undocumented feature of the operating system the file LIFE_STAGES.TXT.SHS would always appear as LIFE_STAGES.TXT. This is because the Shell Scrap files have a Registry entry 'NeverShowExt' which prevents *Explorer* from showing its extension. By using the WScript.Shell object the worm sets another undocumented Registry value (AlwaysShowExt) to ensure that all .TXT files have their extensions show so things will not look out of the ordinary. As a final touch the icon information for .SHS is updated to that of .TXT so even the minor difference between these two file type's icons are removed.

Once running, the worm masquerades as a joke file about the various stages of life for men and women. It accomplishes this with the help of the Scripting.FileSystemObject and the WScript.Shell, dropping a real .TXT file and launching it. The result of this is an impressive set of mimicry. The user sees LIFE_STAGES.TXT in the folder, then launches it and sees LIFE_STAGES.TXT open in *Notepad* or whatever program is associated with .TXT. While reading the message the script is busy doing its malicious work. And what ambitious work it is.

This worm runs the gamut of wormly tricks. It sends itself using the Outlook.Application object, varying the subject line and marking the message for 'delete after send' so as to cover its tracks. It infects IRC programs and transmits itself via these programs. It enumerates network drives and infects them. The most devious of all, it moves the REGEDIT.EXE program into the C:\Recycled folder (updating the association and icon for .REG). Due to the special way *Explorer* handles the Recycled folder it is very difficult, if not impossible, for the average user to undo this trick unaided. The worm is targeted only at *Windows 9x* machines since it does not check that this folder exists (it does not under *NT* or *2000*).

Another new trick was the attempt to obfuscate the code. This took two forms. First, string constants (although not all of them) were encrypted. This makes it difficult to follow the code. It might slow down variants, but not for long. It is a step above the VBS/LoveLetter form where anyone with a text editor could make a variant, no experience necessary. However, it is not difficult and this will be overcome. The second form of obfuscation was in the coding style (or lack thereof) itself. It is reminiscent of the old FORTRAN days and naming conventions. Some of the code does not appear to be VBScript as is shown in Figure 15. It is interesting to note that the method calls to the objects stand out as islands of context against the noise of the code.

```
UQ OE(D('NzCpdvnfmst'))
UQ OE(D('Oqphqbnt'))
Set A=P.CreateTextFile(K(E(1),D('TDBMQFH-UAT')),True)
```

```
 . . .
A.Close
P.CopyFile K(E(1),D('TDBMQFH-
UAT')),K(Left(E(0),3)&D('QFDZDKFC'),D('QDZDKCAM-CBS'))
```

Figure 15: VBS/Stages.A obfuscation

A FIREWALL FOR SCRIPTS

> *fire·wall (fir wôl) n.*
>
> *A fireproof wall used as a barrier to prevent the spread of fire.*
>
> *Computer Science. Any of a number of security schemes that prevent unauthorized users from gaining access to a computer network or that monitor transfers of information to and from the network.*
>
> *Webster's* New World Dictionary

Firewalls have been used in the networking context for decades. A similar approach can be taken towards the scripting technology in *Microsoft Windows* operating systems. By restricting access to the objects that serve as the source of their power we can render these scripts powerless.

COM – The Plumbing That Makes It Work

As stated earlier, script gains power by the use of objects. COM, the Common Object Model, is how these objects are activated. Dynamic Link Libraries (DLLs) provided a level of abstraction between programs and libraries, allowing the underlying implementation to change dynamically. However, this abstraction was limited. A program needed to know where the DLL was located in order to load it. The program needed fairly detailed information regarding the interface.

To further abstract functionality from implementation *Microsoft* came up with COM. With COM, an application needed to know only an object's name or identifier to make use of it.

Due to the fact that scripting is an interpreted language, to be able to programmatically speak with these objects, the interpreter has to be able to determine what methods and properties a given COM object supports on the fly-and be able to interface with them. The IDispatch interface provides this support – GetIDsOfNames returns a list of names and IDs, and parameter types. Those IDs are then passed into the Invoke procedure along with the parameters to execute the method.

Only scripting access to the object is likely to go through the Invoke method, since the amount of setup required is significant and direct calling to the methods is much easier from a compiled language. Given this, we can be relatively certain that anyone calling an object via the Invoke method is doing so from script. Couple this with the fact that accessing the Scripting.FileSystemObject or the WScript.Shell object from an EXE would be silly. Why call an object when you have the Win32 subsystem?

Now we have an area on which to focus our intervention, and thus we have the spot to build our firewall: Invoke.

Heuristic Approach To Firewalls

All methods of all objects called via script must pass through that object's Invoke procedure. At this point we have complete knowledge, before the fact, of the call. We know all the files or Registry keys that are going to be manipulated. We know the network shares being mapped. We know the folders being enumerated. We can use all this information, as well as past history, to form a picture of what the script is doing. Policy can be formulated to describe allowable activity. When a script attempts to do anything that is outside the policy, it is blocked.

Microsoft's security zones are an attempt to protect via policy. Scripts from the Internet were allowed to do one thing; scripts running from the local machine had more flexibility. VBS/BubbleBoy, JS/Kak and VBS/LoveLetter all circumvented this policy by dropping a file onto the user's local machine.

Some products exist now that scan the script source looking for the objects I have discussed. This approach, while it would detect 100% of the malicious scripts seen to date, suffers from two weaknesses. First, it will false positive on good scripts. Corporations which want to use script will suffer many complaints. This will lead to the consequences discussed earlier.

Second, it can be fooled by a number of possible obfuscation techniques. Since no malware using these obfuscation techniques yet exists, I will not go into detail as to how this is accomplished. I will report that a programmer, who learned the scripting language that same day, wrote a proof-of-concept script in several hours. This script used all of the objects scanned for. Yet none of the text-based scanners caught it.

The script encoder provided by *Microsoft* also defeats scanners. Moreover, many of the attachment scanners fail since the extensions are .VBE and .JSE not .VBS and .JS. This is likely to be the next direction these worms will take. As the VBS/Stages.A worm showed us, practically all files would need to be scanned for script. *Microsoft* integrated the scripting so tightly with the OS that one can never be sure which file extension next will be used to launch script.

One key advantage to a heuristic-based scanner is its ability to run at the gateway. It is most desirable to intercept these worms on their first arrival where the cost of dealing with them is lowest. Heuristic-based systems can be written in a platform-neutral manner allowing them to process the attachments on the mail server.

Run-Time Approach To Firewalls

The key advantage to a run-time approach is that no matter what type of obfuscation is employed the object must be instantiated before it can be used. It makes no difference what route is taken. Moreover, since it is activated at run-time the parameters can be examined and to some degree intent can be deduced. This is true also for the vehicle the virus writer chooses to launch the script. It does not matter if the file was .CHM, .VBS, .HTA or .SHS. For the script to do damage it must use one of the objects described in this paper. When this happens the run-time firewall will be there to intercept the call.

When an object's Invoke interface is intercepted, we can determine what method of the object is being called and what parameters are being passed. These parameters will now be in final form regardless of how the script derived them. Directory paths retrieved directly from objects or

pieced together from other strings will arrive as wholly formed strings. For a file copy operation we will know both the source and the destination. We can determine if a network drive mapping is by name or IP address.

Sophisticated rules can be established to govern acceptable behaviour. This can reduce or eliminate false positives. For example, you might allow a script to enumerate the address book. You might allow a script to send mail. However, you might prohibit a script that attempts to do both. You might allow enumeration of the temporary folder but not other key folders. You might allow network drive mappings if the name is in the form \\MachineName while prohibiting the \\www.xxx.yyy.zzz nomenclature.

False Positives or the Software that Cried Wolf

A major impediment to acceptance of proactive software is it propensity for false positives. Presenting users with false alarms can lead to two adverse effects. The first is that the user will get annoyed and disable the software. Protection only works if it is running at the time of attack. The second ill effect is that users get into the habit of dismissing the warning prompt. Since the number of real attacks is an extremely small percentage of the number of intercepts, the odds are that when an alert comes up, it is not real.

After a number of these the user becomes anesthetized and reflexively answers the prompt. When the real threat arrives, it is let through out of habit. Protection that is dismissed at the time of attack is useless. *Outlook* and *Outlook Express* have warning dialogs about executing attachments received via email. Every person who was struck by VBS/LoveLetter, answered ‘Open it’ to the question:

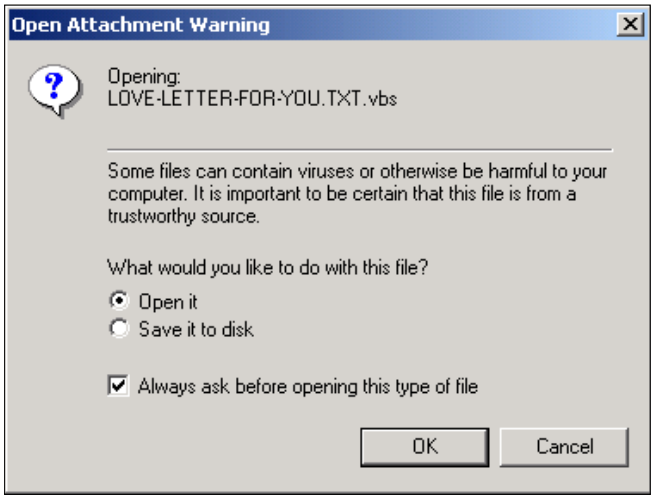

Figure 16: Outlook Express warning dialog ignored by thousands

It is easy to come up with a system that stops all malicious programs. It is easy to come up with system that generates no false positives. The first stops everything and the second stops nothing. Neither is a solution. A balance must be struck between a sufficient degree of protection and intelligence that has a high degree of confidence that when a prompt is presented a threat is present. Under these circumstances, a user will be placed on sufficient guard to respond correctly to the prompt. And what do we do about the corporate script that uses the same objects, many times in the same ways, to accomplish legitimate work?

What do we do about Good Scripts?

Microsoft provided scripting capability for a reason, though some will dispute that decision. Script is extremely useful for performing tasks on machines. Administrators who want to automate tasks can write scripts to perform nearly any action on their user's machines. One script rolled out to an organization can save countless hours. Organizations come to rely on this mechanism. Removing it is simply not an option for them.

The only difference between a good script and a bad script is intent. Intent is very difficult to deduce. It is easy to imagine an administrator writing a script to manipulate some files on a machine, set a Registry key to indicate it has run, and then email a confirmation. It is easy to imagine (since we have seen it) a worm that manipulates files, sets some Registry keys, and sends email. Indeed, a worm whose intention is merely to spread may look much less suspicious than an administrative script. The line between good and bad can be very faint.

Signatures and Scripts

Another way to address the false positive issue is to implement a signature system for 'good' scripts. This would allow a user or an administrator to certify that script be allowed to access the dangerous objects. This system could be extended to use a public/private key pair and hash to validate the script has not been altered.

A further extension would be to encrypt the script code and decrypt it on the fly. The scripting engines themselves are COM objects and can be 'intercepted' in the same way as the other objects are. This technique serves a dual purpose by hiding code from the user. No longer is the script released in source form. Manipulation of the script is not possible.

There may be resistance to this solution because it will require that scripts must be submitted to a central authority for signing. Any modification would require resubmission. This could become quite tedious and some users might resist it. However, if an organization continues to utilize scripting and is struck again with an outside attack, the security needs may eventually outweigh the convenience needs.

CONCLUSION

The continued release of script-based worms must inevitably lead to one of two ends. Either corporations and users will be forced to remove scripting capabilities completely in order to protect themselves, or a solution will be adopted that will prevent the bad external scripts and allow the good internal scripts. The former is the easiest to implement as most draconian solutions are. This is tantamount to surrender. The ramifications of such a solution may make it very costly, particularly in large, managed systems.

The Solonian route is to construct a scripting firewall that allows us to reap the benefits of scripting, while protecting us from rogue elements. Protection and convenience always exist in a ying-yang relationship. The firewall can be built in such a way to obtain the desired balance. The implementations will change from site to site. This too is good. The more variation that exists in organizations, the harder it is to devise broad-based attacks.

It is possible to 'roll-your-own' security with scripting. If one is very familiar with the *Windows* Registry and the manner in which the different elements of the scripting engines work together you can 'rename' the dangerous objects. For instance, the Scripting.FileSystemObject could be Scripting.MyFileSystemObject. Attempts to instantiate the Scripting.FileSystemObject would fail because that object would not exist. The same techniques could be applied to the scripting languages themselves, running .MVB files instead of .VBS. These changes would not be without some degree of risk. *Windows 2000* particularly attempts to protect its files from tampering and may work to counteract renaming attempts.

Several versions of the firewall technology are available today. They all provide complete protection from the scripts we have seen to date. Some will work against future attacks; others will fail. Innovation from both sides will continue. The basic architecture of the scripting model, however, will inevitably tilt the balance towards the defenders. All attackers must come through the same gates, and the defenders will ultimately hold the keys.

ENDNOTES

1 http://msdn.microsoft.com/scripting/jscript/doc/jsfsoTOC.htm

2 http://msdn.microsoft.com/scripting/default.htm?/scripting/windowshost/doc/wsObjWscript.htm

3 http://msdn.microsoft.com/scripting/windowshost/doc/wsObjWshNetwork.htm

4 http://msdn.microsoft.com/library/techart/msdn_movs105.htm

5 http://msdn.microsoft.com/scripting/default.htm?/scripting/scriptlets/doc/letcreatetypelib.htm

6 Data provided by *SARC* submission database.

7 Data provided by *SARC* submission database. April statistics are not available.

8 http://www.cnn.com/2000/ASIANOW/southeast/05/14/philippines.computer.ap/index.html

9 http://www.sarc.com/avcenter/venc/data/vbs.loveletter.a.html (13/06/2000)